Hrvoje Džapo, Vedran Podobnik, Tomislav Pribanić, Marija Seder i Mile Šikić

Informatika

Zagreb, studeni 2017

Sveučilište u Zagrebu
Vojni studijski programi

Sadržaj

1 OSNOVE RAČUNALA 1

1.1 Povijesni razvoj računala 1

1.1.1 Računalne generacije 1

1.2 Građa računala 9

1.3 Binarno zapisivanje podataka 13

1.4 Binarna aritmetika 21

1.4.1 Zbrajanje i oduzimanje 21

1.4.2 Množenje i dijeljenje 26

1.5 Prikaz broja u modulu i komplement broja 28

1.6 Kodiranje binarnim riječima 32

1.6.1 Dekadski kodovi 33

1.7 Otkrivanje i ispravljanje pogrešaka u nizovima bitova 40

1.7.1 Zaštita pomoću zalihosti kodni riječi 40

1.7.2 Zaštita paritetom 42

1.7.3 Hammingovi kodovi 45

2 RAČUNALA I PROGRAMIRANJE 51

2.1 Računala 51

2.2 Programska podrška 52

2.2.1 Operacijski sustav 54

2.3 Pisanje i prevođenje softvera 59

2.4 Tipovi podataka 61

3 OSNOVE PROGRAMIRANJA KROZ PROGRAMSKI JEZIK C 63

3.1 Jednostavan C program 63

3.2 Tipovi podataka u programskom jeziku C 65

3.3 Operacije nad podacima 69

3.3.1 Aritmetički operatori 69

3.3.2 Relacijski operatori 72

3.3.3 Logički operatori ... 74
3.3.4 Operatori nad bitovima ... 74
3.3.5 Kontrola toka podataka ... 76
3.4 Polja, pokazivači i dinamička alokacija memorije ... 83
3.4.1 Polja ... 83
3.4.2 Pokazivači ... 85
3.4.3 Dinamička alokacija memorije ... 87
3.4.4 Polja, pokazivači, reference i dinamička alokacija memorije u jeziku C++ ... 88
3.5 Funkcije ... 90
3.5.1 Deklaracija i definicija funkcije ... 91
3.5.2 Lista argumenta ... 93
3.5.3 Definicija i deklaracija varijabli ... 94
3.5.4 Preopterećenje funkcija ... 95
3.6 Ulazno izlazni tokovi ... 97
3.7 Tokovi podataka u jeziku C ... 99
3.7.1 Standardni tokovima podataka ... 99
3.7.2 Rad s vlastitim tokovima podataka ... 100
3.7.3 Tokovi podataka u jeziku C++ ... 107
4 ALGORITMI I PROGRAMI ... 110
4.1.1 Analiza složenosti algoritama ... 110
5 PROCESI PROGRAMSKOG INŽENJERSTVA ... 121
5.1 Proces oblikovanja programske potpore ... 121
5.2 Modeli procesa programskog inženjerstva ... 124
5.3 Agilni procesi ... 130
6 OBJEKTNO ORIJENTIRANO PROGRAMIRANJE ... 137
6.1 Objektno orijentirana paradigma ... 137
6.2 Razredi i objekti ... 141
6.3 Apstrakcija ... 146

6.4 Enkapsulacija ... 147
6.5 Nasljeđivanje ... 156
6.6 Polimorfizam ... 162

7 UML MODELIRANJE ... 168

7.1 Modeliranje programske potpore pomoću UML-a ... 168
7.2 Dijagram razreda (engl. *class diagram*) ... 171
7.3 Dijagram objekata (engl. o*bject diagram*) ... 180
7.4 Dijagram obrazaca uporabe (engl. *use case diagram*) ... 183
7.5 Interakcijski dijagrami (engl. *interaction diagrams*) ... 188
7.6 Dijagram stanja (engl. *statechart diagram*) ... 192
7.7 Dijagram aktivnosti (engl. *activity diagram*) ... 194
7.8 Dijagram komponenata (engl. c*omponent diagram*) ... 197
7.9 Dijagram razmještaja (engl. *deployment diagram*) ... 199

8 GRAFOVI ... 201

8.1 Osnovni pojmovi ... 201
8.2 Vrste grafova ... 202
8.3 Povezanost grafova, šetnje i putovi ... 203
8.3.1 Primjena Eulerovog i Hamiltonovog puta ... 205
8.4 Reprezentacije grafova ... 207
8.4.1 Lista susjedstva ... 207
8.4.2 Matrica susjedstva ... 209
8.4.3 Matrica incidencije ... 210
8.5 Poveznica između grafa i stabla ... 212
8.6 Obilazak grafa ... 215
8.6.1 DFS obilazak vrhova grafa ... 215
8.6.2 BFS obilazak vrhova grafa ... 219
8.6.3 Razapinjuće stablo ... 222
8.7 Topološki sort ... 223
8.7.1 Nalaženje najkraćeg puta između vrhova grafa ... 226

9 MREŽE RAČUNALA I KOMUNIKACIJSKE MREŽE 229

9.1 Uvod u mreže računala 230

9.2 Vrste računalnih mreža 234

9.2.1 Topologija računalnih mreža 234

9.2.2 Rasprostranjenost računalnih mreža 237

9.2.3 Vrsta informacija i način razmjene informacija u računalnim mrežama..... 238

9.2.4 Pokretljivost korisnika i namjena računalnih mreža..... 240

9.3 Arhitektura mreže 241

9.4 Komunikacijski protokol 245

9.5 Internetski protokolni složaj 246

9.5.1 OSI RM sloj 1: Fizički sloj i prijenosni medij 247

9.5.2 OSI RM sloj 2: Sloj podatkovne poveznice i lokalna mreža..... 250

9.5.3 OSI RM sloj 3: Mrežni sloj 251

9.5.4 OSI RM sloj 4: Transportni sloj 252

9.5.5 OSI RM sloj 5: Sjednički sloj 254

9.5.6 OSI RM sloj 7: Prezentacijski sloj 254

9.5.7 OSI RM sloj 7: Aplikacijski sloj 255

10 LITERATURA 259

1 Osnove računala

U ovom poglavlju naveden je kronološki razvoj računala. Također, navedene su i opisane osnovne gradivne jedinke računala. Poglavlje završava pregledom brojevnih sustava s posebnim naglaskom na binarni brojevni sustav.

1.1 Povijesni razvoj računala

Povijest računala pokriva razvoj od vrlo jednostavnih uređaja koji služe kao pomoć pri računanju matematičkih operacija do koncepta modernog računala. Rani mehanički uređaji koji su pomagali ljudima u računskim operacijama nazivani su računalnim strojevima ili kalkulatorima, dok je glavni dio stroja, operator za računanje, nazvan računalom. Takva računala su napredovala kroz vrijeme tako što su od početne funkcionalnosti rada sa znamenkama odnosno cjelobrojnim vrijednostima, unaprijeđene sa složenim operacijama nad kontinuiranim vrijednostima (primjerice, duljina skale, napon, zakret osovine i sl). Tako su nastala analogna računala. Nakon niza tehnoloških proboja, kao što je pojava tranzistora, integriranih sklopova, digitalna računala zamijenila su u potpunosti analogna računala. Cijena računala s vremenom pada te se pojavljuju osobna računala 90-tih godina, a pametni telefoni i tableti postaju sveprisutni dio civilizacije nakon 2000. godine.

Koncept modernog računala stvorio je u devetnaestom stoljeću engleski matematičar Charles Babbage (1791.–1871.). Počeo je raditi 1834. godine na uređaju kojeg je nazvao analitičkim strojem. Tadašnja tehnologija dovela ga je do neuspjeha da ostvari svoju zamisao jer masovna proizvodnja uniformnih mehaničkih dijelova s niskim tolerancijama tada još nije postojala. Tijekom Drugog svjetskog rata pojavila su se prva "prava računala" kada su Englezi stvorili Colossus računalo za dešifriranje njemačkih kriptografskih strojeva koji su kodirali vojne poruke (Enigma, Tunny). Nakon drugog svjetskog rata Amerikanci su preuzeli prvenstvo u tehnologiji, proizvodnji i primjeni računala.

1.1.1 Računalne generacije

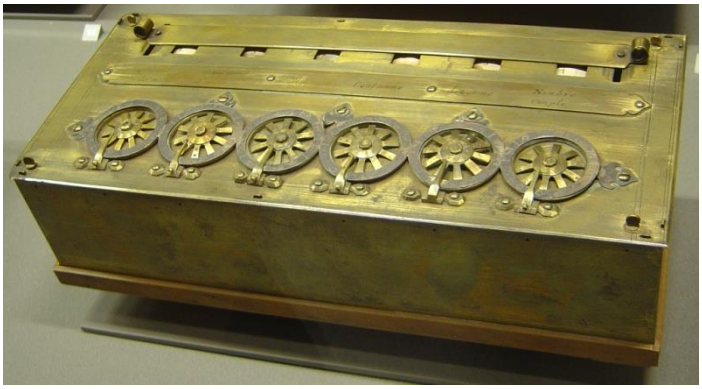
U literaturi se obično spominju četiri generacije računala, koje su zasnovane na tehnologiji koja je u danom vremenskom razdoblju bila korištena za izgradnju temeljnih sklopova računala [1]. Te generacije su proširene s tzv. nultom generacijom računala, bez koje sami

razvoj računala ne bi bio moguć, a koja ne obuhvaća današnju modernu definiciju računala. Peta generacija je nadodana kao najnoviji trend u razvoju računala koja obuhvaća najnovije algoritme i kombinaciju više računala u koncept umjetne inteligencije.

- Nulta generacija računala – mehanički strojevi za računanje (1644. – 1944.);
- Prva generacija računala – elektronička računala s elektronskim cijevima (1945. – 1955.);
- Druga generacija računala – tranzistor kao građevna komponenta (1955. – 1965.);
- Treća generacija računala – integrirani sklopovi (1965. – 1980.);
- Četvrta generacija računala – sklopovi vrlo visokog stupnja integracije (1980. – danas);
- Peta generacija računala – koncept umjetne inteligencije.

Nulta generacija računala - mehanički strojevi za računanje (1644. – 1945.)

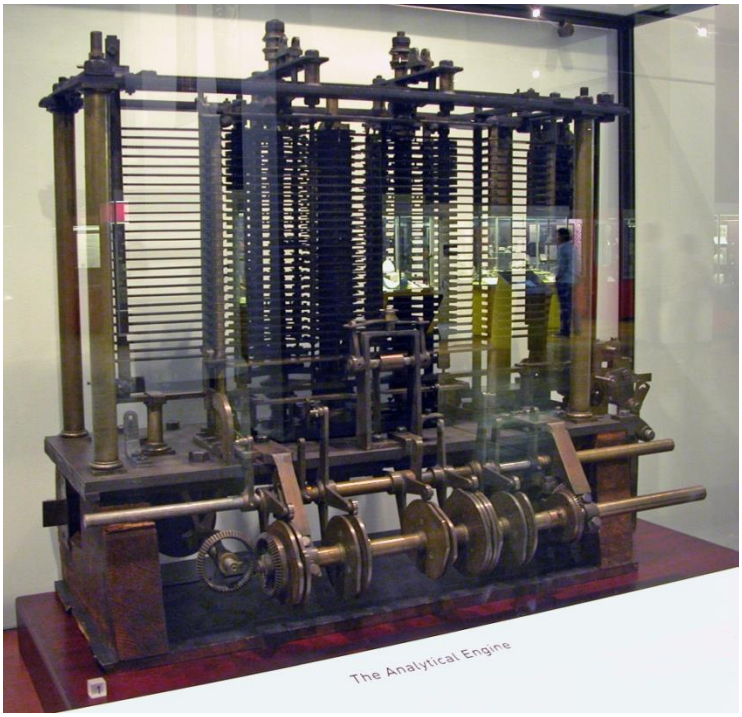
Mehanički kalkulatori su jedna od ranih ideja da bi dugotrajne matematičke operacije trebali obavljati strojevi a ne ljudi. Pojavu mehaničkih kalkulatora valja obilježiti Pascalovim izumom 1642. koji je nakon 50 prototipova nazvan Paskalovim kalkulatorom (Slika 1.1). Paskalov kalkulator sastojao se od mehaničkoga sustava kotačića čiji je položaj određivao znamenku. Mogao je obavljati operacije zbrajanja i oduzimanja te množenja i dijeljenja pomoću ponavljajućih operacija zbrajanja odnosno oduzimanja.



Slika 1.1 Pascalov kalkulator. Preuzeto iz [2].

Povijest spremanja podataka započela je pojavom bušenih kartica, koje je izumitelj Joseph-Marie Jacquard 1801. godine patentirao za pohranu uzoraka i dezena koji su se izrađivali u

tkanini pomoću tkalačkih stanova. Ovaj koncept spremanja informacija iskoristio je Charles Babbage u svom analitičkom stroju. Charles Babbage, kojeg se naziva "ocem računala", zamislio je da stroj čita dva niza bušenih kartica – jedan s naredbama a drugi s podacima, a rezultat izračunavanja buši se na treći niz kartica. Njegov analitički stroj bio je mehaničko računalo opće namjene. Sastojao se od oko 25,000 dijelova, težio 15 tona i bio 2.4 m visok (Slika 1.2). Imao je memoriju, jedinicu za računanje, ulaznu jedinicu (čitač bušenih kartica) i izlazne jedinice (pisač i bušač kartica).



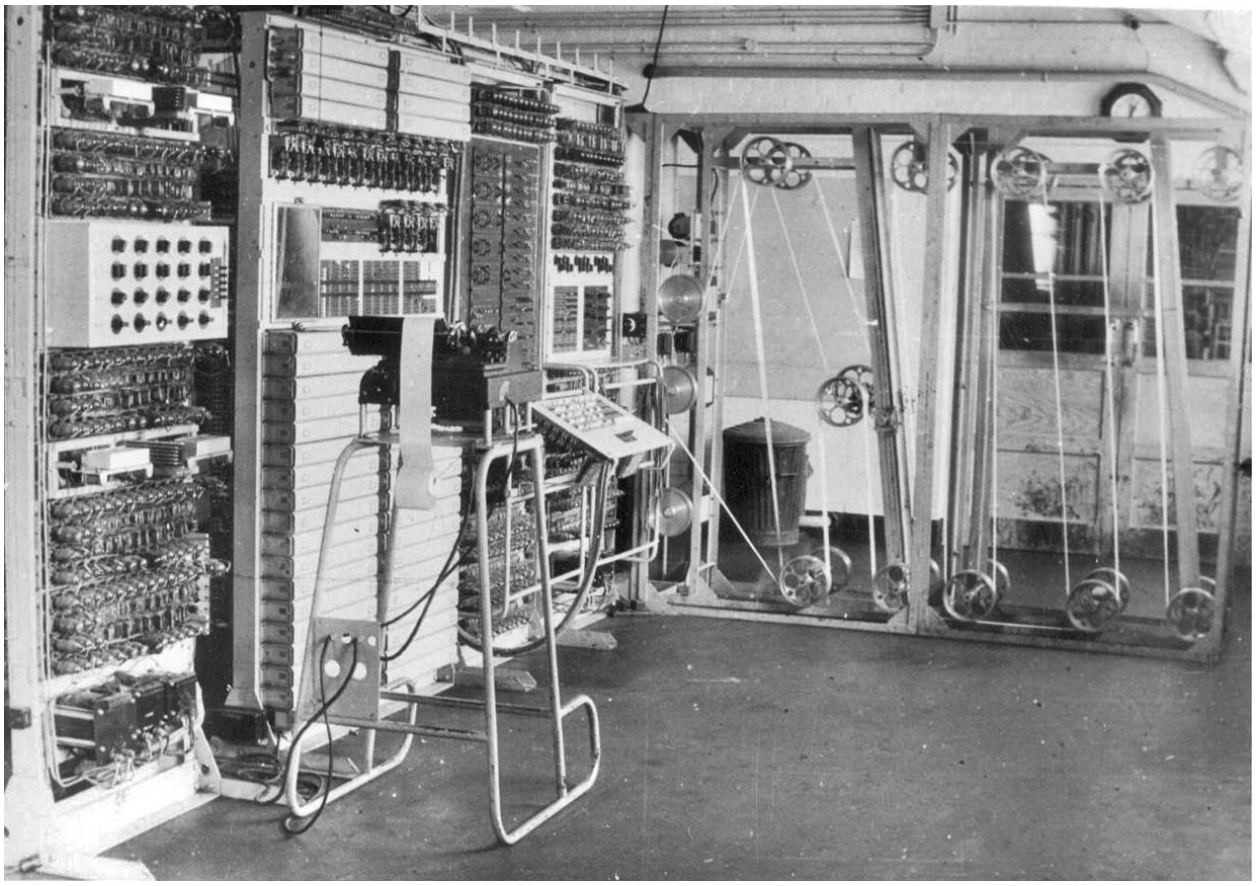
Slika 1.2 Babbageov analitički stroj. Preuzeto iz [3].

Stroj zbog nemogućnosti tadašnje tehnologije nije nikad završen. Dalje je potreba za razvojem računala stavila svoj fokus na vojne ciljeve. Nijemci su u drugom svjetskom ratu napravili Enigmu - elektromehanički stroj za šifriranje vojnih poruka. Enigma se sastojala od tipkovnice na kojoj se pisala poruka, a pritiskom na pojedino slovo zakrenuo bi se sustav s tri rotirajuća diska te bi zasvijetlilo šifrirano slovo. Šifriranu poruku je bilo moguće dešifrirati ako bi se znala početna postavka diskova i spojeva električnih žica na ploči s utičnicama i preklopkama koja se nalazila na stražnjoj strani Enigme. Britanci su 1939. godine razvili elektromehaničko računalo za dešifririranje Enigme nazvano Bombe, kojeg je začetnik bio matematičar Alan Turing (1912. – 1954.). Računalo je radilo dva do tri zadatka simultano

tako što je provjeravalo moguće postavke Enigme izvođenjem lanaca logičkih dedukcija implementiranih električki. Većina mogućnosti je završavalo kontradikcijom, a nekoliko preostalih opcija provjeravalo se ručno. Nijemci su također razvili nekoliko telegrafskih kriptiranih sustava, potpuno različitih od Enigme. Tako je stroj Lorenz SZ 40/42 korišten za vojne komunikacije na visokoj razini, kojeg su Britanci zvali "Tunny". Kao napad na Tunnyja, 1941. britanski matematičar Max Newman počeo je specificirati računalo Colossus.

Prva generacija računala - elektronička računala s elektronskim cijevima (1945. – 1955.)

Colossus je prvo elektroničko programirljivo računalo stvoreno 1944. Koristilo je veliki broj elektronskih cijevi. Imalo je kao ulaz papirnatu traku i moglo ga se podesiti da izvodi razne logičke operacije nad podacima (Slika 1.3).



Slika 1.3 Colossus računalo. Preuzeto iz [4].

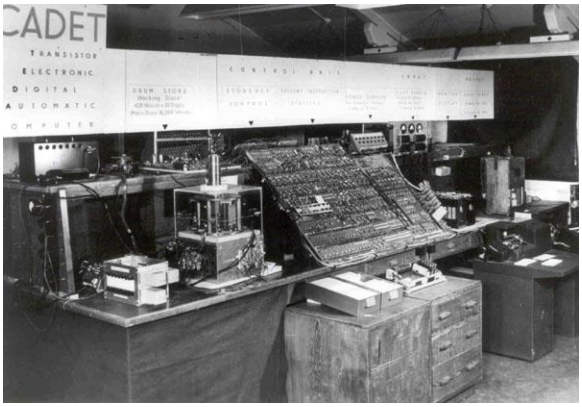
Međutim, Colossus nije bio Turing–kompletan, svojstvo koje je nazvano prema britanskom matematičaru Alanu Turingu koje govori zadovoljava li računalo osnovni koncept Turingovog stroja. Teoretsku osnovu Turingovog stroja postavio je Turing 1936. godine te dao prve specifikacije elektroničkog digitalnog računala 1945. godine. Pokazao je da Turingov stroj može izvesti bilo koji matematički izračun ako se on može prezentirati algoritmom. Pojam Turing–kompletan odnosi se na stroj koji može izvršiti bilo koji program koji se može pohraniti u njegovu memoriju. Amerikanci su 1946. godine izgradili računalo ENIAC (engl. *Electronic Numerical Integrator and Computer*) vrlo slično Colossusu, ali mnogo brže i prilagodljivije računalo. Bilo je Turing–kompletno i moglo je izračunati bilo koji problem koji se mogao zapisati u njegovu memoriju. Kao i kod Colossusa, program na ENIAC–u definirao se kabelima i preklopkama. Kada bi se napisao program, morao se postaviti u stroj ručnim postavljanjem utikača i prekidača. ENIAC je imao 18000 elektronskih cijevi, 1500 releja, 30 tona i mogao je izvršavati 3 operacije korjenovanja u sekundi, što je neopisivo sporo po današnjim standardima. Matematički savjetnik razvojne grupe koja je izgradila ENIAC, John von Neumann (1903.–1957.), 1945. godine postavio je osnovnu logičku strukturu računala s memoriranim programom prema kojoj računalo mora imati:

i) ulazni dio pomoću kojeg se može unositi proizvoljni broj operanada i naredaba;
ii) memoriju u koju se mogu unositi i dohvaćati operandi i naredbe;
iii) aritmetičko-logičku jedinicu koja može obavljati sve potrebne aritmetičke i logičke operacije s operandima i pohranjivati rezultate u memoriju;
iv) izlazni dio preko kojeg se rezultati na nekom izlaznom sredstvu predaju korisniku;
v) upravljačku jedinicu koja može interpretirati naredbe dobivene iz memorije i upravljati s ostalim dijelovima računala.

Ova osnovna struktura poznata je kao Von Neumannova struktura i na njoj su zasnovana sva moderna računala. Iako je ova struktura u suštini slična Turingovoj strukturi koja je teoretski čvršće postavljena, Von Neumannova struktura je dobila na popularnosti.

Druga generacija računala - tranzistor kao građevna komponenta (1955. – 1965.)

Godine 1947. Walter Brittain i Willaim Shockley izumili su tranzistor, koji je počeo polako istiskivati elektronske cijevi iz upotrebe. Tranzistori su omogućili izradu kompaktnijih sustava. U isti fizički obujam, uz istu disipiranu snagu, bilo je moguće smjestiti uređaje koji imaju mnogo puta veću procesnu moć. Istodobno s razvojem tranzistora razvile su se i magnetske memorije, koje su bile brže, pouzdanije i jeftinije od prethodne akustičke memorije. Prvo potpuno tranzistorsko računalo CADET (Slika 1.4) napravili su Britanci 1955. u laboratoriju Harwell. CADET je mogao kontinuirano raditi više od 80 sati. Međutim, postojao je problem pouzdanosti rada računala koja se poboljšala pojavom bipolarnih tranzistora. Prvo komercijalno tranzistorsko računalo Metrovick 950 napravila je britanska elektrotehnička tvrtka Metropolitan-Vickers 1956. godine. Šest računala Metrovick 950 bila su u radu oko 5 godina. Nakon Metrovicka 950 oko jednu trećinu svjetskog tržišta preuzima tvrtka IBM sa svojim IBM 1401. IBM je instalirao više od 10 000 ovih računala između 1960. i 1964. godine.

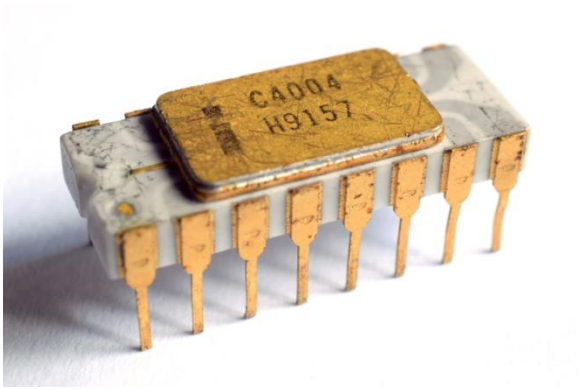


Slika 1.4 Prvo tranzistorsko računalo – CADET. Preuzeto iz [5].

Treća generacija računala - integrirani sklopovi (1965. – 1980.)

Integrirani sklopovi niskog stupnja integracije sastavni su dijelovi treće generacije računala. Pojavljuju se integrirane poluvodičke memorije koje zamjenjuju magnetske memorije. Integrirani sklop je skup velikog broja tranzistora na komadiću silicija odnosno germanija. Prvi integrirani sklop razvili su neovisno 1959. godine američki inženjeri Jack Kirby i Robert Noyce. Razvoj integriranog sklopa doveo je do izuma mikroprocesora. Prvi mikroprocesor

bio je 1971. godine 4-bitni Intel 4004 [6] (Slika 1.5) i sadržavao je samo CPU (engl. *Central Processor Unit*).



Slika 1.5 Intelov procesor 4004. Preuzeto iz [6].

Kasnije su mikroprocesori sadržavali sve komponente računala. Već sljedeće 1972. godine izlazi 8-bitni mikroprocesor Intel 8008, a 1976. godine 8-bitni mikroprocesor Intel 8048 sadrži ulazne i izlazne pristupe, radnu memoriju (RAM) i programsku memoriju (ROM). Godine 1978. pojavljuje se 16-bitni mikroprocesor Intel 8086, a njemu konkurira Motorola 68000, koji je interno 32-bitni mikroprocesor. Godine 1975. Steve Jobs i Steve Wozniak pokreću Apple Computer i stvaraju prvo stolno računalo Apple I. Računala treće generacije imaju operacijski sustav, koji omogućuje vremensku podjelu programskog rada, odnosno omogućuje višeprogramski i višeprocesni rad. Pojavljuju se ulazno/izlazni dijelovi računala tipkovnice i monitori.

Četvrta generacija računala – sklopovi vrlo visokog stupnja integracije (1980. – danas)

Daljnji napredak elektroničke tehnologije vidi se u povećanju broja sklopova koji se mogu smjestiti na jediničnu površinu silicijske pločice odnosno čip. Mikroračunala, koja na jednoj pločici imaju sve bitne dijelove računala, glavni su nositelj ove četvrte generacije. Prvi potpuni 32-bitni mikroprocesor objavio je Intel 1981. godine. Te je iste godine IBM izdao osobno računalo s Intelovim 8088 mikroprocesorom. Godine 1984. izlazi Apple Macintosh s grafičkim korisničkim sučeljem. Prvo prijenosno računalo (engl. *laptop*) izlazi 1981. godine – Osborne 1 (**Slika 1.6**).

Slika 1.6 Prijenosno računalo Osborne 1. Preuzeto iz [7].

Prvo prijenosno računalo s ekranom u boji izlazi 1988. godine – Compaq. Godine 2008. izlazi najtanje prijenosno računalo Macbook Air. Godine 2011. izlazi najtanje prijenosno računalo s operacijskim sustavom Windows – Dell XPS 15Z. Od razvoja pametnih telefona valja navesti iPhone kao prvi telefon s ekranom osjetljivim na dodir koji je izašao 2007. godine te HTC Dream zasnovan na Googleovom operacijskom sustavu Android koji je izašao 2008. godine.

Tehnološki napredak razvoja računala bio je toliko značajan da je suosnivač tvrtke Intel Gordon Moore u svom istraživanju 1965. godine postavio pravilo da se broj tranzistora na integriranom sklopu udvostručuje svake dvije godine te postavio projekciju ovog pravila na barem još sljedećih 10 godina. Ovo pravilo naziva se Mooreov zakon i ne opisuje nikakav fizikalni zakon već projekciju odnosno pogled razvoja tehnologije. Mooreova predikcija pokazala se točnom nekoliko decenija, a povezuje se i s memorijskim kapacitetima, senzorima te čak brojem i veličinom piksela u digitalnim kamerama. Očekivano je da će Mooreov zakon u jednom trenu prestati vrijediti, međutim to još nije jasno kada. U 2015. godini izmjereno je da se rast usporio na svake dvije i pol godine.

Valja istaknuti neke današnje trendove (2017. godine): Microsoft održava Windowse, a zadnja verzija je Windows 10; operacijski sustav Linux postaje sve popularniji; IBM prestaje raditi mala računala – Lenovo je kupio njihov posao 2005. godine; HP je najveći proizvođač osobnih računala (PC-a), a drugi je Lenovo; Apple je najveći igrač za osobna računala za "kreativno tržište"; Google je najveći igrač za internetska rješenja.

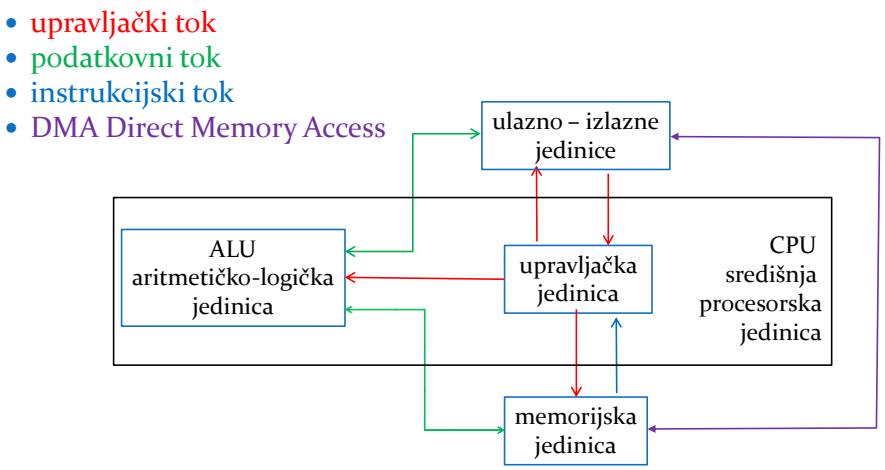
Peta generacija računala – koncept umjetne inteligencije

Pojava paralelne obrade više instrukcija pokazuje mogućnost umnažanja snaga više CPU-a. Cilj je pete generacije razviti uređaje koji imaju sposobnost učenja i samoorganizacije. U ovu generaciju se svrstavaju istraživačke grane robotika i nanotehnologija. U klasičnom pristupu računalo će riješiti problem točno na način kako je programirano, bez ikakvoga unaprjeđenja, alternativnih rješenja, mogućih prečica, ili greški u izvođenju programskog koda. Računalni programi koji uče i mijenjaju se tokom izvođenja dio su modernog koncepta umjetne inteligencije i strojnog učenja. Neki primjeri su prepoznavanje glasa, fonta, prijevoda i on-line reklamiranje.

1.2 Građa računala

Hardver (engl. *hardware*) računala čine fizički dijelovi računala, kao što su monitor (engl. *monitor*), tipkovnica (engl. *keyboard*), pohrana podataka, grafička kartica (engl. *graphic card*), zvučna kartica (engl. *sound card*), matična ploča (engl. *motherboard*). Nasuprot hardveru, softver (engl. *software*) je niz instrukcija koji se može pohraniti na hardver i kojeg pokreće hardver. Softver upravlja hardverom tako što hardver izvršava bilo koju naredbu koja je zapisana u softveru. Kombinacija hardvera i softvera čini korisni računalni sustav.

Osnova i predložak arhitekture svih modernih računala je Von Neumannova struktura (Slika 1.7). Prema Von Neumannovoj strukturi računalo je podijeljeno na središnju procesorsku jedinicu (CPU), koja se sastoji od aritmetičko logičke jedinice (skr. ALU, od engl. *Arithmetic Logic Unit*) i upravljačke jedinice, memorije (engl. *memory*) u koju se pohranjuju podaci i instrukcije te ulazno-izlazne jedinice (engl. *input-output unit*). Detaljnije objašnjenje ovih komponenata može se naći u [8].



Slika 1.7 Von Neumannova struktura računala.

Kućište računala

Kućište računala obuhvaća većinu komponenata računalnog sustava. Ono daje mehaničku podršku i zaštitu unutarnjih elemenata kao što su matična ploča, vanjska pohrana podataka i moduli napajanja. Također upravlja i usmjerava protok rashladnog zraka preko unutarnjih elemenata. Dodatno upravlja elektromagnetskim međudjelovanjima koje računalo zrači i štiti unutarnje elemente od elektrostatičkog pražnjenja. Postoje dizajni kao što je Appleov iMac koji u istom kućištu imaju i grafički prikaz. Prijenosna računala zahtijevaju kućišta koja imaju zaštitu od udaraca. Novije izvedbe prijenosnih računala i tableti imaju ekran osjetljiv na dodir.

Modul napajanja

Modul napajanja (engl. *power supply*) pretvara izmjenični naponski izvor gradske mreže u nisko-naponski istosmjerni signal koje koriste interne komponente računala. Prijenosna računala mogu raditi na ugrađenoj bateriji po nekoliko sati.

Matična ploča

Matična ploča (engl. *motherboard*) glavna je komponenta računala. Na njoj se nalazi integrirani sklop koji povezuje ostale dijelove računala: CPU, RAM (engl. *Random Access Memory*), vanjske pohrane (CD, DVD, tvrdi disk, itd.) kao i periferiju spojenu na priključcima ili utorima za proširenje.

Centralna procesorska jedinica

Centralna procesorska jedinica (skr. CPU, od engl. *Central Processing Unit*) izvodi glavninu izračuna računala te se stoga naziva "mozgom" računala. Obično se hladi ventilatorom ili rashladnim uređajem zasnovanom na vodi. Većina novijih CPU-ova sadrži grafičku procesorsku jedinicu - GPU (engl. *Graphics Processing Unit*). Brzina takta CPU-a određuje kako će se brzo izvoditi instrukcije i mjeri se u GHz. Tipične vrijednosti danas su između 1 i 5 GHz. Mnoga moderna računala imaju opciju ubrzanja rada CPU-a nauštrb povećanju grijanja CPU-a.

Memorija: RAM

Memorija RAM (engl. *Random Access Memory*) je brza memorija koja pohranjuje program i podatke kojima CPU aktivno pristupa. Svojstvo slučajnog pristupa (engl. *random access*) znači da je potrebno isto vrijeme da se pristupi bilo kojoj memorijskoj lokaciji, odnosno podacima se efikasno pristupa u bilo kojem slučajnom poretku. Ovo je u suprotnosti s uređajima za pohranu podataka kao što je CD ili tvrdi disk, koji čita i piše podatke slijedno te stoga podacima pristupa u istom slijedu kao što su i zapisani. Kao primjer korištenja RAM-a navedimo otvaranje mrežnog preglednika (engl. *web browser*). Tada se koristi memorija tako što se pohranjuju potrebni podaci za efikasni prikaz stranice sve dok se mrežni preglednik ne zatvori. RAM obično dolazi u veličinama od 2GB, 4GB, 8GB, ili 16GB, međutim može biti i veća.

Memorija: ROM

Memorija ROM (engl. *Read-Only Memory*) memorija je koja se samo može čitati i koja pohranjuje BIOS (engl. *Basic Input-Output System*) koji se pokreće kod pokretanja računala (engl. *booting*). Također, BIOS uključuje tvorničke postavke računala i upravljanje napajanjem. Novije matične ploče koriste UEFI (engl. *Unified Extensible Firmware Interface*) umjesto BIOS-a.

Sabirnice

Sabirnice (engl. *bus*) povezuju CPU s raznim unutarnjim komponentama računala. Preko njih se prenose podaci, adrese i upravljački signali.

Grafička kartica

Grafička kartica (engl. *graphic card*) ili video kartica obrađuje računalnu grafiku. Postoje moćne grafičke kartice koje mogu izvršavati zahtjevne zadatke kao što su intenzivne video igrice.

Uređaji za pohranu podataka

Uređaj za pohranu podataka može pohraniti podatke i otpakirati podatke. Može pohraniti podatak privremeno ili trajno. Može biti vanjski ili unutarnji uređaj na računalu, a može biti server. Pohrana podataka je ključna funkcija i osnovna komponenta računala.

Fiksna pohrana

Podaci koje pohranjuje računalo mogu biti na raznim medijima. Tvrdi disk se nalazi u starijim računalima i vrlo je niske cijene a velikog kapaciteta. SSD (engl. *Solid-State Drivers*) je noviji i mnogo brži medij, međutim trenutno znatno skuplji nego tvrdi disk.

Uklonjiva pohrana

Da bi se podaci prenijeli od računala do računala koriste se USB pohrane ili optički diskovi. Računala danas sve rjeđe imaju uređaj za čitanje optičkog diska, a sva nova računala imaju barem jedan USB ulaz.

Ulazni i izlazni moduli

Ulazni i izlazni moduli obično su postavljeni izvan kućišta računala. Ulazni moduli omogućavaju korisniku da upišu informaciju u računalni sustav, ili da upravljaju njegovim izvođenjem. Većina osobnih računala imaju miša i tipkovnicu, a na prijenosnim računalima se češće koristi ugrađena dodirna podloga (engl. *touchpad*). Ostali ulazni moduli mogu biti mrežna kamera, mikrofon, komandna palica (engl. *joystick*), skener, itd. Izlazni moduli prikazuju informaciju u čovjeku prilagođenom obliku. Takvi moduli mogu biti printer, zvučnici, monitor, projektor.

1.3 Binarno zapisivanje podataka

Prema Von Neumannovoj arhitekturi računalo se sastoji od četiri glavne komponente: aritmetičko logičke jedinice, upravljačke jedinice, memorije i ulazno-izlaznih modula. Ove komponente povezane su sabirnicama. Unutar svake komponente nalaze se milijuni malih električnih sklopova koji se mogu uključiti ili isključiti poput električne sklopke. Svaki sklop predstavlja jedan bit kao binarnu znamenku, koji može biti "1" kada je sklop uključen ili "0" kada je sklop isključen. Ovi sklopovi su organizirani u logička vrata tako što jedan ili više sklopova mogu utjecati na stanje jednog ili više drugih sklopova. Izvedba ovih sklopova te način funkcioniranja digitalnih sustava detaljno su objašnjeni u [8].

Bit

Bit ili binarna znamenka (engl. *binary digit*) je osnovna jedinica informacije u računalnim i digitalnim komunikacijama. Binarna znamenka može imati samo dvije vrijednosti koje se osim "0" i "1" mogu prezentirati i s logičkim vrijednostima "true" i "false", aktivnim stanjima "on" i "off", ili bilo kojim drugim dvoznačnim vrijednostima. Što ove dvije vrijednosti predstavljaju fizikalno stvar je konvencije i moguće je imati različita pridruživanja unutar istog uređaja ili programa. Bit se može pohraniti u fizički sustav kao dva moguća stanja. To mogu biti dva stabilna stanja bistabila, dva položaja električne sklopke, dvije različite naponske razine u sklopu, dvije različite razine intenziteta svjetlosti, itd. U uređajima koji koriste pozitivnu logiku, vrijednost "1" predstavlja se s višom razinom napona od razine napona koju predstavlja "0". Vrijednosti napona variraju ovisno o faktoru starenja komponenata i osjetljivosti na šum. Primjerice, u tranzistor-tranzistor logici (skr. TTL, od engl. *transistor-transistor logic*) vrijednosti "0" i "1" definirane su vrijednostima napona ne višom od 0.4 V, odnosno ne manjom od 2.6 V. U ranim razdobljima računala koje se naziva nultim generacijskim razdobljem, kao što je doba Babbagova analitičkog stroja, bit je bio predstavljen pozicijom mehaničke poluge ili zupčanika, dok je kod bušenih kartica bit bio predstavljen prisustvom ili odsustvom rupe na kartici. Prvi elektronički uređaji s diskretnom logikom kao što su regulatori za dizala i semafore, imali su bitove predstavljene stanjima električnih releja koji su mogli biti otvoreni ili zatvoreni. Kad su releje zamijenile elektronske cijevi, mogućnosti predstavljanja bita postale su mnogostruke: tlak u stupcu

živinih para, električni naboj, itd. Pojavom magnetskih memorija bit je bio predstavljen magnetskim polaritetom, ili promjenom polariteta. Ova reprezentacija bita se i danas koristi u kreditnim karticama s magnetskom trakom. U današnjoj poluvodičkoj memoriji RAM, dvije vrijednosti bita predstavljene su s dvije razine naboja pohranjenog na kondenzatoru. U memoriji ROM vrijednosti bita predstavljene su prisustvom ili odsustvom staze vodljivosti.

Grupiranje bitova

Više bitova mogu se predstaviti na više načina. Grupiranje bitova u digitalnim sustavima radi prikaza podataka čini nekoliko osnovnih jedinica: grupa od 8 bitova čini oktet ili bajt (engl. *byte*); grupa od 16 bitova čini riječ (engl. *word*) – u današnjim računalima broj bitova u riječi iznosi 32 ili 64. Povijesno gledano oktet se koristio da bi predstavio jedan znak teksta i zbog toga je bio korišten kao osnovni element za adresiranje u mnogim računalnim arhitekturama. Danas je uvriježeno UTF-8 višebajtovno kodiranje tekstualnog znaka.

Prijenos i obrada podataka

Bitovi se prenose jedan po jedan u serijskom prijenosu, ili u većem broju odjednom u paralelnom prijenosu podataka. Operacije nad pojedinim bitovima obrađuju bitove jedan po jedan. Brzina prijenosa podataka obično se mjeri u broju bitova koji se mogu prenijeti u jedinici vremena (primjerice, bit/s ili kbit/s ili Mbit/s, itd.).

Prikaz podataka

Zbog toga što u računalu sve informacije možemo predstaviti samo pomoću bitova, koriste se tzv. binarni kodovi, odnosno binarni brojevi. Primjerice, zamislimo sljedeće pridruživanje brojeva slovima tako da je slovu "A" pridružen 1, slovu "B" pridružen 2, ..., slovu "Ž" pridružen 30, tada bi tekst "ŽABA" bio zapisan kao niz brojeva 30,1,2,1. Ako ovu poruku želimo poslati preko mreže s druge strane mora postojati isti algoritam pridruživanja, odnosno moramo se dogovoriti oko standarda kodiranja. U binarnom svijetu računala postoje samo vrijednosti 0 i 1. Grupa bitova naziva se binarnim kodom i može predstavljati tekst, procesorsku instrukciju ili nekakav spremljeni podatak. Binarni kod pridružuje kombinaciju binarnih znamenki (bitova) svakom tekstualnom znaku, instrukciji ili podatku. Primjerice, binarni kod od 8 bitova može predstaviti 256 mogućih kombinacija te stoga predstaviti

različite elemente. Tako standardni ASCII kod koji je 8 bitna riječ predstavlja 256 različitih tekstualnih znakova, npr. znak malo slovo "a" predstavljen je binarnom riječi 01100001, što je u dekadskom sustavu broj 97. U računarstvu i telekomunikacijama, koriste se različite metode kodiranja podataka, a kao glavni primjer uzmimo pretvaranje tekstualnih znakova u niz bitova. Metode kodiranja mogu biti fiksne duljine binarnih kodova (broj bitova u riječi) ili različite duljine. U kodiranju koje koristi fiksnu duljinu binarnih kodova, svaki znak ili znamenka predstavljen je nizom bitova uvijek iste duljine, a taj se niz preveden u binarni broj obično prikazuje u kodnoj tablici u oktalnom, dekadskom ili heksadekadskom zapisu. Postoji mnogo skupova znakova i za njih mnogo različitih metoda kodiranja. Primjerice, u kodu cp1250 slovo "ž" prikazano je binarnom riječi 10011110 (dekadskim brojem 158) dok je u kodu ISO 8859-2 to isto slovo prikazano binarnom riječi 10111110 (dekadskim brojem 190). Većina se (standardnih) znakova kodiraju na isti način, ali kod prikaza specijalnih znakova, kao što su "čćđšž", moramo paziti da postavimo odgovarajuće kodiranje.

Brojevni sustavi

Binarno brojanje je slično brojanju u bilo kojem drugom brojevnom sustavu. Počevši od najdesnije znamenke kao prve, brojanje se nastavlja kroz sve druge znamenke u rastućem redoslijedu. U dekadskom sustavu postoje 10 mogućih vrijednosti znamenki – od 0 do 9. Brojanje počinje od najmanje značajne znamenke (najdesnije), koja se najčešće zove i prva znamenka, ili znamenka jedinica. Kada se sve znamenke izredaju, najmanje značajna znamenka postavi se u 0, a prva se sljedeća znamenka poveća za 1, znamenka desetica. Proces brojanja izgleda kako slijedi:

000, 001, 002, ... 007, 008, 009, (najdesnija se znamenka postavi u nulu, a uveća se znamenka lijevo)

010, 011, 012, ...

...

090, 091, 092, ... 097, 098, 099, (najdesnije se dvije znamenke postave u nule, a uveća se prva sljedeća znamenka)

100, 101, 102, ...

Binarno brojanje odvija se na isti način, ali uzimajući u obzir da postoje samo dvije vrijednosti znamenki – 0 i 1. Stoga, već nakon 1 dolazi 0, a uveća se prva sljedeća znamenka:

0000,

0001, (najdesnija se znamenka postavi u nulu, a uveća se znamenka lijevo)

0010, 0011, (najdesnije dvije znamenke postave se u nule, a uveća se prva sljedeća znamenka)

0100, 0101, 0110, 0111, (najdesnije tri znamenke postave se u nule, a uveća se prva sljedeća znamenka)

1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 ...

Ako bismo prethodno pobrojane binarne brojeve redom brojali na prirodan način (u dekadskom sustavu) dobili bismo pretvorbu iz binarnog sustava u dekadski: prvi navedeni broj 0000 je 0, 0001 je 1, 0010 je 2, ..., a zadnji navedeni broj 1111 je 15. Formula kojom direktno možemo odrediti dekadski ekvivalent binarnog broja bez brojanja jest sljedeća: svaka znamenka predstavlja rastuću potenciju broja 2, gdje je najdesnija znamenka 2^0, sljedeća znamenka lijevo do nje 2^1, sljedeća znamenka 2^2, i tako dalje. Zbrajanjem svih potencija od broja 2 nad znamenkama s vrijednosti 1 dobivamo dekadski ekvivalent. Kao primjer, binarni broj 100101 se pretvara kako slijedi:

$100101 = [(1) \times 2^5] + [(0) \times 2^4] + [(0) \times 2^3] + [(1) \times 2^2] + [(0) \times 2^1] + [(1) \times 2^0]$

$100101 = [1 \times 32] + [0 \times 16] + [0 \times 8] + [1 \times 4] + [0 \times 2] + [1 \times 1]$

100101 = 37 u bazi 10

Tablica 1.2 prikazuje brojevne sustave u bazi 2, 3, 8, 10 i 16 s mogućim vrijednostima znamenki, dok Tablica 1.1 daje prvih 16 brojeva u dekadskom, binarnom, oktalnom i heksadekadskom sustavu.

Tablica 1.1 Prvih 16 brojeva u različitim brojevnim sustavima.

dekadski	binarni	oktalni	heksadekadski
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Tablica 1.2 Primjeri brojevnih sustava i njihove znamenke.

baza B	brojevni sustav	znamenke sustava (B)
2	binarni	0,1
3	ternarni	0,1,2
8	oktalni	0,1,2,3,4,5,6,7
10	dekadski	0,1,2,3,4,5,6,7,8,9
16	heksadekadski	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Pretvorbe brojeva u različitim sustavima

Da bi se pretvorio broj iz baze 10 (dekadski sustav) u bazu 2 (binarni sustav), broj se dijeli s dva. Ostatak je najmanje značajan bit. Kvocijent se dalje dijeli s dva i njegov ostatak je sljedeća znamenka lijevo od najmanje značajnog bita. Proces se ponavlja sve dok kvocijent ne postane 0. Niz ostataka čini binarni broj. Primjerice, dekadski broj 345 je binarni 101011001 prema sljedećem postupku:

345 : 2 = 172 ostatak 1

172 : 2 = 86 ostatak 0

86 : 2 = 43 ostatak 0

43 : 2 = 21 ostatak 1

21 : 2 = 10 ostatak 1

10 : 2 = 5 ostatak 0

5 : 2 = 2 ostatak 1

2 : 2 = 1 ostatak 0

1 : 2 = 0 ostatak 1

Pretvorba iz baze 2 u bazu 10 je obrnuti proces. Počevši od najznačajnijeg bita (najlijevijeg) i početne vrijednosti rezultata 0, prethodna vrijednost rezultata se množi s dva, a sljedeći bit se

dodaje za računanje nove vrijednosti rezultata. Procedura traje sve dok se ne iscrpe svi bitovi. Ovu proceduru možemo zapisati i drugačije, tako što svaku znamenku množimo s potencijom broja dva ovisno o tome na kojoj je udaljenosti znamenka od najmanje značajne znamenke.

Binarni broj 1 0 0 1 0 1 0 1 1 0 1

Procedura $((((((((((0\times2+\mathbf{1})\times2+\mathbf{0})\times2+\mathbf{0})\times2+\mathbf{1})\times2+\mathbf{0})\times2+\mathbf{1})\times2+\mathbf{0})\times2+\mathbf{1})\times2+\mathbf{1})\times2+\mathbf{0})\times2+\mathbf{1}$

Dekadski broj $1\times2^{10}+0\times2^{9}+0\times2^{8}+1\times2^{7}+0\times2^{6}+1\times2^{5}+0\times2^{4}+1\times2^{3}+1\times2^{2}+0\times2^{1}+1\times2^{0}$ = 1197

Pretvaranje binarnog broja u heksadekadski je znatno jednostavnije budući da je baza 16 četvrta potencija od broja 2. Stoga treba binarni broj grupirati po četiri bita i pretvarati zasebno svaku grupu u jednu heksadekadsku znamenku. Ako binarni broj nema dovoljno znamenki da se tvore grupe od četiri bita, treba nadodati vodeće nule na početak niza brojeva da bi se dobila grupa od 4 bita.

Binarni broj 1010010 = 0101 0010 (nadodana jedna vodeća nula) = 52 u bazi 16

Binarni broj 11011101 = 1101 1101 = DD u bazi 16

Pretvaranje heksadekadskog broja u binarni radi se supstitucijom svake heksadekadske znamenke s četiri bita koji moraju biti nadopisani vodećim nulama da popune četiri mjesta.

Heksadekadski broj 3A = 0011 1010 u bazi 2

Heksadekadski broj E7 = 1110 0111 u bazi 2

Pretvaranje heksadekadskog broja u dekadski radi se isto kao i pretvaranje iz binarnog u dekadski, osim što se ovdje radi o potencijama broja 16.

Heksadekadski broj C0E7 = $12\times16^{3}+0\times16^{2}+14\times16^{1}+7\times16^{0}$ = 49,383 u bazi 10

Logičke operacije nad bitovima

Osim standardnih algebarskih operacija nad binarnim brojevima (zbrajanje, oduzimanje, množenje, dijeljenje) postoje i logičke operacije. Logičke operacije nad jednim ili više bitova u računalima izvedene su pomoću dioda ili tranzistora i mogu ostvariti sve logičke funkcije definirane Booleovom algebrom. Booleova algebra grana je matematike koja koristi varijable

koje mogu imati samo dvije vrijednosti, istinu (1) i laž (0). Osnovne operacije Booleove algebre su I, ILI i NE.

I (presjek), označava se $x \wedge y$ (može i x AND y), zadovoljava $x \wedge y = 1$ ako vrijedi $x = y = 1$, a $x \wedge y = 0$ inače.

ILI (unija), označava se $x \vee y$ (može i x OR y), zadovoljava $x \vee y = 0$ ako vrijedi $x = y = 0$, a $x \vee y = 1$ inače.

NE (komplement), označava se $\neg x$ (može i NOT x, ili !x), zadovoljava $\neg x = 0$ ako je $x = 1$, a $\neg x = 1$ ako je $x = 0$.

Alternativno, vrijednosti $x \wedge y$, $x \vee y$, i $\neg x$ mogu se izraziti tablicama istinitosti kako slijedi: Tablica 1.3 prikazuje vrijednosti funkcija I i ILI za sve moguće kombinacije vrijednosti x i y, dok

Tablica **1.4** prikazuje vrijednosti funkcije NE za dvije moguće vrijednosti x.

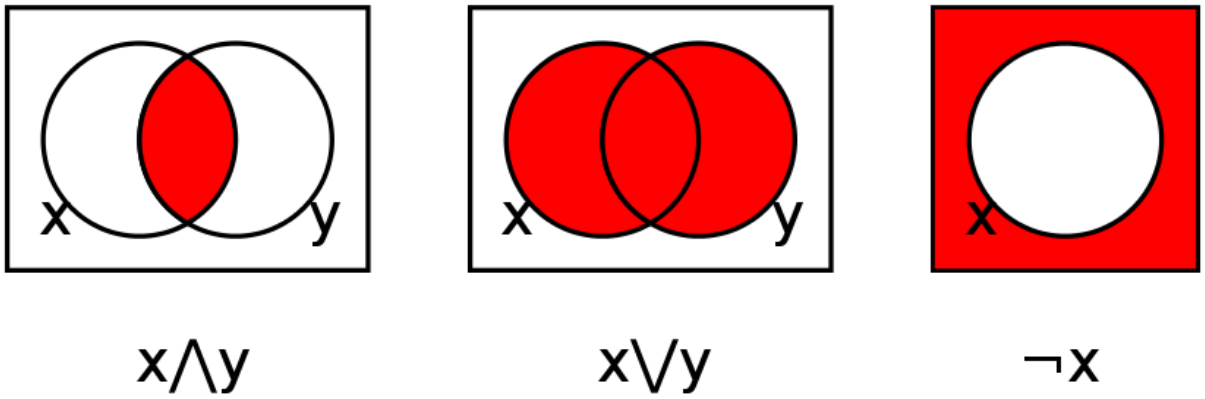
Tablica 1.3 Tablica istinitosti za funkcije I i ILI.

x	y	$x \wedge y$	$x \vee y$
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	1

Tablica 1.4 Tablica istinitosti za funkciju NE.

x	$\neg x$
0	1
1	0

Ove se operacije mogu ilustrirati i operacijama nad skupovima odnosno Vennovim dijagramima (Slika 1.8). Operacija I predstavlja presjek dva skupa, operacija ILI predstavlja uniju dva skupa, a operacija NE predstavlja komplement skupa.



Slika 1.8 Vennovi dijagrami prikaza operacija I, ILI i NE. Preuzeto iz [10].

1.4 Binarna aritmetika

1.4.1 Zbrajanje i oduzimanje

Aritmetičke operacije su jedne od ključnih funkcija digitalnog sustava (računala), a u sklopu njih je zbrajanje temeljna operacija. S obzirom na uobičajeni rad i prikaz brojeva u binarnome sustavu, pravila zbrajanja su relativno jednostavna:

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ \underline{+0} & \underline{+1} & \underline{+0} & \underline{+1} \\ 0 & 1 & 1 & 1|0 \end{array} \tag{1}$$

Od četiri moguće kombinacije rezultata prve tri daju na izlaz jedan bit dok u četvrtom slučaju se dešava pojava i pojava prelijeva. Zbog toga se pored rezultata sume *S* uvodi i eksplicite dodatna veličina za prijenos *C* pa se zbrajanje neka dva bita *A* i *B* opisuje tablicom zbrajanja (Tablica 1.5).

Tablica 1.5 Tablica zbrajanja dvije binarne znamenke

A	*B*	*S*	*C*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

U računalu je moguće imati sklopovlje koje bi zbrajalo samo dvije znamenke kao što prikazuje Tablica 1.5. Ali tada je nužno je imati i odgovor na pitanje što kad se pojavi prijenos *C* na nekom binarnom mjestu. Prilikom zbrajanja na papiru, pojava prijenosa te dodavanje na sumu dviju znamenki na većem binarnom mjestu, obavlja se u glavi. Drugim riječima zbrajaju se tri broja. Očigledno s sklopovljem koje neposredno uzima samo dvije znamenke na ulazu to nije izvedivo. Alternativa je započeti zbrajati parove znamenki, kao da nema preljeva, no svaku pojavu preljeva pamtiti (u posebnom registru), tako da se zapiše na jedno brojno mjesto više od onoga na kojem je nastalo. Na rezultat koji se dobije nastavlja se zbrajanje s brojem zabilježenim kroz pojavu preljeva. Postupak se nastavlja sve dok se pojavljuje preljev. Izraz (2) pokazuje postupka na primjeru zbroja dva broja dekadskog brojevnog sustava, A=467 i B=38. Nakon prvog koraka za sumu S pamtimo zbrojeve po parovima znamenki brojeva A i B dok eventualno pojavu preljeva pamtimo na posebnom mjestu (registru) C. Međurezultata nakon prvog koraka glasi S=495, C=10. Nastavljamo zbrajati po parovima znamenki brojeva, kao i u prethodnim koracima, što daje novi međurezultat S=405, C=100. U sljedećem koraku S postaje S=505, dok za C=0 više nema zapisane pojave preljeva, čime je postupak završen.

$$
\begin{array}{rrrrl}
 & 4 & 6 & 7 & : A \\
+ & & 3 & 8 & : B \\
\hline
 & 4 & 9 & 5 & : S \\
+ & & 1 & 0 & : C \\
\hline
 & 4 & 0 & 5 & : S \\
+ & 1 & 0 & 0 & : C \\
\hline
 & 5 & 0 & 5 & : S \\
 & & & & : C
\end{array}
\qquad (2)
$$

Za mnoge česte operacije u računalima opravdano je napraviti sklopovlje koje će u jednom koraku obaviti neku zadaću. U slučaju zbrajanja to bi značilo sklopovlje koje na ulaz prima ne dva nego tri podatka/broja. Dva broja se odnose znamenke *i*–tog binarnog mjesta, a treći je podatak upravo mogući prijenos s nižeg binarnog mjesta *i-1*. Takav sklop treba imati također i dva izlaza, jedan za sumu i drugi za prijenos (preljev) na više brojevno mjesto. Tablica 1.6 pokazuje 8 mogućih kombinacija na ulazu takvoga sklopa zajedno s 8 korespondentnih izlaza.

Tablica 1.6 Tablica zbrajanja tri binarne znamenke.

A	B	C_{i-1}	S	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Oduzimanje je također vrlo važna operacija koja se može provesti ili korištenjem komplementa broja i preko operacije (sklopovlja) zbrajanja ili je moguće oduzimanje

direktno provesti operacija. U posljednjem slučaju osnovu čine pravila zbrajanja u binarnome sustavu. Slično kao i kod zbrajanja, tri od četiri slučaja daju relativno jednostavne izlazne rezultate za razliku (diferenciju) *D*. Međutim četvrti slučaj zahtjeva uvođenje dodatnog izlaza koji bilježi pojavu zajma ili posudbe *Z* s više brojnog mjesta. Tablica 1.7 prikazuje za slučaj oduzimanja samo dvije binarne znamenke *A* i *B*, četiri moguća izlazna rezultata za diferenciju *D* i zajam *Z*. I ovdje se očituje ograničenje što sklopovlje bi operaciju oduzimanja trebalo obavljati u nekoliko koraka jer za ulazne vrijednosti ne uzima u obzir i mogući zajam s prethodnog brojevnog mjesta.

Odgovarajući sklop za neposredno oduzimanje triju znamenki treba na ulazu uzeti o obzir sve tri znamenke, od čega su dvije od njih minuend i suptrahend na nekoj i-toj brojevnoj poziciji, a treća znamenka je zajam s susjednog binarnog mjesta *i-1*.

Tablica 1.8 pokazuje 8 mogućih kombinacija na ulazu takvoga sklopa zajedno s 8 korespondentnih izlaza. Vrijedno je primijetiti kako je rezultat sume *S* (Tablica 1.6) jednak rezultatu diferencije *D* (Tablica 1.8) a, što se potencijalno može iskoristiti kod optimizacije izbora sklopovlja za zbrajanje i oduzimanje.

Tablica 1.7 Tablica oduzimanja dvije binarne znamenke

A	*B*	*D*	*Z*
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Tablica 1.8 Tablica oduzimanja tri binarne znamenke.

A	B	Z_{i-1}	D	Z_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Stupci za prijenos *C* (Tablica 1.6) i zajam Z (Tablica 1.8) nisu jednaki. Ipak pažljiviji promatrač će primijetiti da su prva četiri retka stupca preljeva *C* jednaka zadnja četiri retka stupca zajma *Z*, tj. da su zadnja četiri retka stupca preljeva *C* jednaka prva četiri retka stupca zajma *Z*. Takvo obilježje je također iskoristivo kod optimizacije sklopovlja za zbrajanje i oduzimanje gdje se odgovarajućim ulaznim signalom, koji definira operaciju zbrajanja ili oduzimanja nad ulaznim bitovima, može iskoristi gotovo isto sklopovlje za obje aritmetičke operacije.

1.4.2 Množenje i dijeljenje

Za neki par multiplikanda *M* i multiplikatora *N* najjednostavnije rješenje za množenje bilo bi koristiti sklopove za zbrajanje te uzastopno zbrojiti *N* puta multiplikand, tj. *M* puta multiplikator. Nedostatak ovakvog rješenje je relativno dugo izvršenje čitave operacije množenja za veće *M* i/ili *N*. Alternativa je implementaciju množenja realizirati na osnovu vrlo jednostavne tablice množenja za binarni sustav:

$$\begin{aligned} 0 \cdot 0 &= 0 \\ 0 \cdot 1 &= 0 \\ 1 \cdot 0 &= 0 \\ 1 \cdot 1 &= 1 \end{aligned} \tag{3}$$

Sljedeći dalje pravila množenja kao i na papiru, množenje multiplikanda *M* i multiplikatora *N* je tada definirano kroz dobivanje parcijalnih produkata i njihovu sumaciju.

$$\begin{array}{ccccccc} & & M & \cdot & N & & \\ 1 & 0 & 1 & \cdot & 1 & 1 & 0 \\ \hline 1 & 0 & 1 & & & & \\ & 1 & 0 & 1 & & & \\ & & 0 & 0 & 0 & & \\ \hline 1 & 1 & 1 & 1 & 0 & & \end{array} \tag{4}$$

Neposredna sklopovska provedba izraza (4) se može napraviti na više načina. Primjerice za četverobitni multiplikand $M=m_3m_2m_1m_0$ i četverobitni multiplikator $N=n_3n_2n_1n_0$ možemo parcijalna množenja i sume izraziti kao:

$$M \cdot N = M \cdot (n_3 \cdot 2^3 + n_2 \cdot 2^2 + n_1 \cdot 2^1 + n_0 \cdot 2^0) \tag{5}$$

Množenje potencijom broja 2, gdje ta potencija može biti potencijalno vrlo velika, je sklopovski efikasnije izvesti množenjem samo sa 2^1 prema tzv. Hornerovoj shemi:

$$M \cdot N = \big((M \cdot n_3 \cdot 2 + M \cdot n_2) \cdot 2 + M \cdot n_1\big) \cdot 2 + M \cdot n_0 \tag{6}$$

Sklopovska realizacija Hornerove sheme prema izrazu (6) više stupnjeva zbrajala na čije ulaze se dovode pojedini umnošci $M \cdot n_i$. i posmaknuti izlazi zbrajala prethodnog stupnja. Očigledno ovakvo rješenje za množenje je značajnije brže od ranije spomenutog uzastopnog zbrajanja. Međutim broj ulaza za zbrajalo je određen brojem bitova multiplikanda *M*, a broj

potrebnih zbrajala je određen s brojem bitova mutiplikatora N. Drugim riječima povećanje bitova bilo multiplikanda, bilo mutliplikatora, značajno utječe na složenost sklopa za množenje. Svojevrsno kompromisno rješenje bi uključivalo množenje mutiplikanda s bitovima multiplikatora te posmak bitova takvog umnoška za jedne mjesto te pribrajanje takovih parcijalnih produkata. Za ovo rješenje je nužno imati sklop za posmak bitova, a što je u sklopovskom smislu puno povoljnije nego puno zbrajala i svako od njih s velikim brojem ulaza. Alternativa za množenje je također sklop koji bi za neki par brojeva na ulazu davao direktno produkt binarnih brojeva na izlazu. Međutim takav sklop tipično nema druge svrhe osim množenja pa se često pribjegava ranijem rješenju s zbrajanjem parcijalnih produkata.

Konačno operaciju dijeljenja je moguće provesti također prema pravilima dijeljenja na papiru gdje se divizor podjeli s brojem najvažnijih bita dividenda. Interesantno je kako u odnosu na dekadski sustav postupak je još jednostavniji u binarnom budući da se nakon spuštanja sljedeće znamenke na rezultat dodaje samo 1, ako je divizor manji ili 0, ako je divizor veći. Ipak zbog mogućnosti pojave ostatka dijeljenje je nešto složenije od množenja, pa jednostavni digitalnu sustavi nemaju poseban sklop za dijeljenje već se ono implementira programski. Slično tome se još složenije operacije poput potenciranja, logaritmiranja itd. implementiraju programski ili pomoću preglednih tablica.

1.5 Prikaz broja u modulu i komplement broja

U računalu se pohrana brojeva odvija u registrima gdje se pamte pojedine znamenke nekog broja (Slika 1.9). Očigledno je broj raspoloživih mjesta u registru ograničen na neki konačan broj od *n* mjesta i gdje je taj *n* najčešće definiran duljinom riječi za dotičan računalni sustav.

b_{n-1}	b_{n-2}	b_{n-3}	…	b_1	b_0

Slika 1.9 Shematski prikaz smještaja binarnog broja *b* od *n* znamenki u registar.

Uz pretpostavku binarnog brojevnog sustava čija je baza *B*=2, najveći prikazivi broj u registru s *n* mjesta je $W=B^n-1=2^n-1$, tj. broj različitih stanja koje se može prikazati je $m=B^n=2^n$, gdje se *m* naziva još i modul. Način rada sklopovlja je takav da se u registrima prikazuju (ostaju zapisani) svi brojevi u tzv. modulu. To znači da za neki općeniti broj *a* izražen kao:

$$a = z \cdot m + b \tag{7}$$

gdje je *z* neki cijeli broj, ostaje zapisano u registru samo vrijednost *b*. Proizlazi da ukoliko je bilo $a<=W$ tada će u registru biti stvarno i zapisana vrijednost *a* dok u suprotnome slučaju ostaje zapisan ostatak nakon dijeljenja broja *a* s modulom *m*. Alternativni način zapisa izraza (7) glasi:

$$b = a(mod\ m) \tag{8}$$

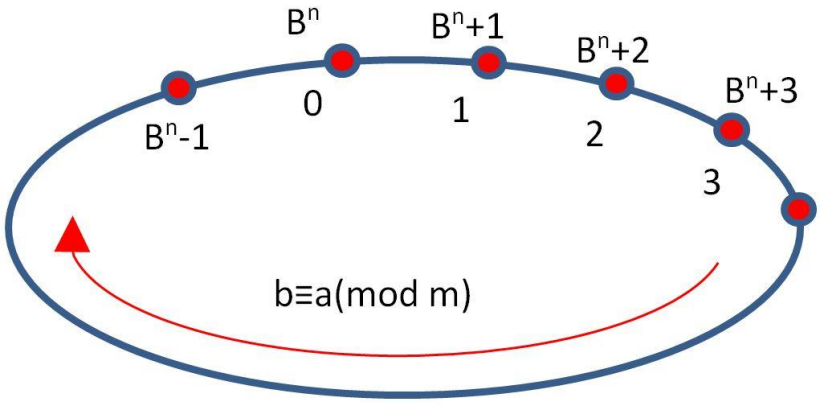
Nerijetko se u matematičkoj terminologiji odnos u gornjem izrazu (8) iskazuje zamjenom znaka jednakosti s znakom kongruencije ≡:

$$b \equiv a(mod\ m) \tag{9}$$

Npr. pretpostavimo da je *m*=7 i vrijednost broja *a*=1. Tada bi sljedeće brojevi dekadskog sustava bili u kongruentnom odnosu

$$\begin{gathered} a \equiv a + k \cdot m, k = \cdots - 2, -1, 0,1, 2\ldots \\ 1 \equiv -13 \equiv -6 \equiv 8 \equiv 15 \end{gathered} \tag{10}$$

Ponekad je za razumijevanje zapisa broja u modulu *m*, za neku bazu *B* brojevnog sustava i uz raspoloživi prikaz mjesta *n*, korisno cijeli princip ilustrirati slikom (Slika 1.10), gdje se dodatno naglašava nemogućnost zapisa broja većeg od B^n-1.



Slika 1.10 Ilustracija zapisa broja u modulu.

U kontekstu prikaza broja u modulu $m=B^n$, definiraju se komplementi brojeva, a dva popularna su tzv. B-ti i (B-1) komplement. Prvo navedeni komplement $\overline{N_B}$ se odnosi na sami modul *m* tj. za broj prikazivih stanja, a za neki broj *N* definiran je:

$$\overline{N_B} = B^n - N \tag{11}$$

Drugo navedeni komplement $\overline{N}$ je za neki broj *N* definiran u odnosu na najveći prikazivi broj *W*:

$$\overline{N} = W - N = B^n - N - 1 \tag{12}$$

U slučaju dekadskog brojevnog sustava B=10, pretpostavke ograničenja na *n*=2 brojna mjesta nekoliko sjedećih primjera pokazuje izračun *10*-tog komplementa:

$$\begin{aligned} \overline{19_{10}} &= 10^2 - 19 = 81 \\ \overline{21_{10}} &= 10^2 - 21 = 79 \\ \overline{77_{10}} &= 10^2 - 77 = 23 \end{aligned} \tag{13}$$

Na analogan način se za binarni sustav može izračunati B=2 komplemenet broja $21_{10}=010101_2$ uz pretpostavku za primjerice *n*=5 binarnih mjesta:

$$\overline{010101_2} = 2^5 - 10101 = 100000 - 10101 = 01011 \tag{14}$$

Pažljivijom analizom binarne reprezentacije polaznog broja i njegovog 2 komplementa može se zaključiti kako je direktno oduzimanje brojeva u binarnom sustavu, u cilju izračun 2-komplementa, obično praktičnije zamijeniti algoritmom koji bi počevši od najmanje značajnog bita polaznog broja, invertirao svaki bit nakon prve 1.

Izraz (15) daje za dekadski sustav primjere izračuna B-1=9 komplementa:

$$\begin{aligned} \overline{19} &= 10^2 - 19 - 10^0 = (10^2 - 1) - 19 = 80 \\ \overline{21} &= 10^2 - 21 - 10^0 = (10^2 - 1) - 21 = 78 \\ \overline{77} &= 10^2 - 77 - 10^0 = (10^2 - 1) - 77 = 22 \end{aligned} \tag{15}$$

Slično tome B-1=1 komplement u binarnom sustavu za isti broj 21_{10}=010101_2, iz primjera (14), bi se nalazio:

$$\overline{010101_2} = 2^5 - 10101 - 1 = 100000 - 10101 = 01010 \tag{16}$$

Analizom izraza (16) uviđa se da se B-1=1 komplement binarnog sustava dobiva jednostavno komplementiranjem početnog niza bitova, a što je naročito pogodno za sklopovsku izvedbu. Zatim temeljem činjenice da su 2 komplement i 1 komplement povezani izrazom:

$$\overline{N_B} = \overline{N} + 1 \tag{17}$$

ranije izložen algoritam oko izračuna 2 komplement može se dodatno pojednostaviti, naročito u kontekstu sklopovske izvedbe. Dakle u prvom koraku se izračuna 1 komplement invertiranjem svih bitova (ne mora se više analizirati pojava 1 na najmanjem značajnom bitu) i jednostavno se na takav međurezultata još samo nadoda 1.

Jedna od najvažnijih motivacija za uporabom komplementa brojeva je korištenje sklopova za zbrajanje i u funkciju oduzimanja, čime se očito pojednostavljuje obavljanje aritmetičkih operacija. Neka su *M* i *S* minuend i suptrahend brojeva čiju razliku *D*=*M*-*S* želimo izračunati, a $\overline{S_B}$ je *B*-ti komplement suptrahenda *S*. Tada vrijedi sljedeće:

$$D = M - S = M - (B^n - \overline{S_B}) = M + \overline{S_B} - B^n$$
$$M + \overline{S_B} = D + B^n \tag{18}$$

Ključno zapažanje koje proizlazi iz izraza (18) je to da je suma minuenda i B-tog komplementa suptrahenda kongruentna s razlikom *D*=*M*-*S* koju se zapravo traži.

$$M + \overline{S_B} = D + B^n$$
$$D \equiv M + \overline{S_B} \tag{19}$$

Drugim riječima dovedemo li na ulaze sklopa za zbrajanje minuend *M* i od suptrahenda *S* njegov B-ti komplement $\overline{S_B}$, temeljem ranije objašnjenog svojstva sklopovskog zapisa i prikaz broja u modulu, ono što će ostati zapisano u registru bit će barem vrijednost kongruentna diferenciji *D* koju tražimo ako ne i sama razlika *D*. Preciznije govoreći postoje dva slučaja s obizorm na karakter sadržaja $M + \overline{S_B}$ koji će biti zapisan u registru: Prvi slučaj podrazumijeva kada je *M*>*S*, čime je pozitivna razlika, *D*>0. Drugi slučaj je obrnuta situacija kada je kada je *M*>*S*, pa imamo negativnu razliku, *D*<0. Za prvi slučaj *D*>0, izraz (19) možemo raspisati kao:

$$M + \overline{S_B} = M + B^n - S = D + B^n = D + W + 1 > W \tag{20}$$

Očigledno da sadržaj koji se nastoji zapisati u registar prilikom pokušaja zbrajanja $M + \overline{S_B}$ je veći od najveće prikazivog broja *W*, pa će nastati preljev (stanje koje je inače moguće detektirati posebnim sklopom). Međutim preljev je samo narušio jednakost ali ne i kongruenciju. Zato onaj ostatak koji će ostati zapisan u registru upravo odgovara diferenciji *D* koju i tražimo. Za prvi slučaj *D*<0, izraz (19) možemo raspisati kao:

$$M + \overline{S_B} = M + B^n - S = D + B^n = D + W + 1 < W \tag{21}$$

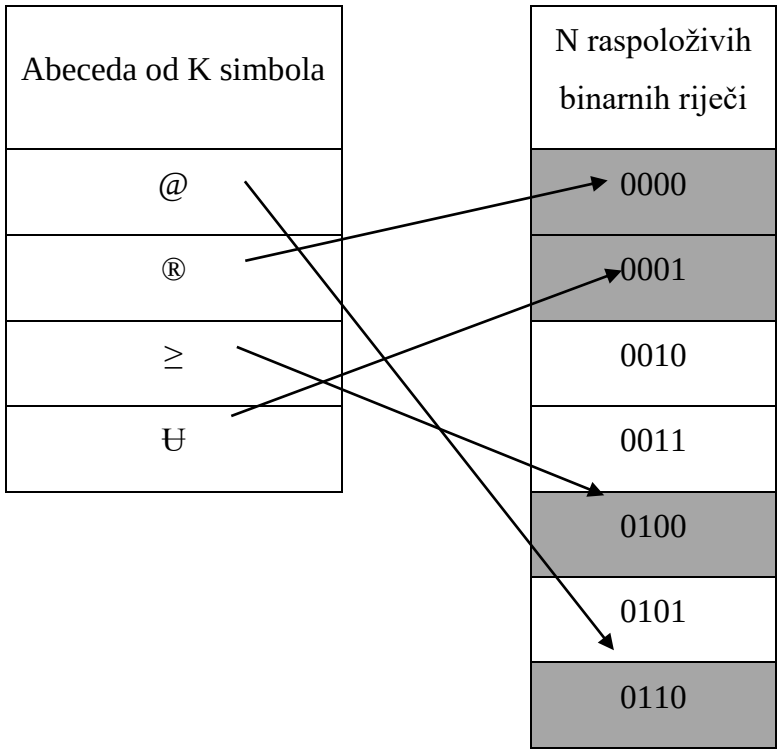
U ovome slučaju jer je D<o očigledno upisujmo sadržaj koji je manji od maksimalno prikazivog broja **W**, pa neće nastati preljev. Međutim s obzirom da je *S*>*M* iznos koji će ostati neposredno ostati u registru zapisan nakon sume $M + \overline{S_B}$ nije onaj koji se stvarno traži. Iznos $M + \overline{S_B}$ odgovara broju koji je točno za D manji od najveće prikazivog broja *W*. Stoga je od prvotnog sadržaja registra potrebno napraviti B-ti komplement (i promijeniti predznak)

da bi se dobio očekivani iznos *D*. Konačno čitav algoritam koji oduzimanje M-S provodi operacijom zbrajanja *M* i uporabom suptrahendovog B-tog komplementa S_B, može se sažeto opisati kao:

- pribrojiti minuendu M komplement suptrahend S_B
- u slučaju preljeva, to je i rezultat
- ako nije došlo do preljeva tada je potrebno komplementirati sadržaj registra te promijeniti predznak

1.6 Kodiranje binarnim riječima

Tablica 1.9 Dodijele kodnih riječi četiri simbola – kodiranje. Odabrane kodne riječi, koje sačinjavaju kod, su u zasjenjenim ćelijama tablice.



Prikaz binarnim znakovima nije ograničen samo na brojeve binarnog sustava već se njima može prikazati proizvoljan sadržaj. S jedne strane to je velika prednost jer omogućava praktički obradu bilo koje vrste informacija digitalnim sustavom (računalom), iako iz samoga

niza 1 i 0 naravno nije jasno o kojem se tipu podatka radi već je potrebno znati to unaprijed. Sami postupak izražavanja simbola (znakova) u binarnom obliku se naziva kodiranje. Nizu bitova kojem je dodijeljeno neko značenje zovemo kodna riječ, a skup takvih kodnih riječi čini određeni *kod*. Abeceda je skup svih oni simbola koje su prikazane kodnim riječima dok se sami elementi abecede (simboli) nazivaju još i znakovima. Inverzni postupak prilikom kojeg iz svake kodne riječi nalazimo pojedini simbol (element abecede) se naziva dekodiranje. Abeceda može npr. predstavljati skup slova, specijalnih znakova, dekadskih znamenki i sl. Tablica 1.9 daje primjer kodirana K=4 simbola s kodnim riječima svaka duljine 4 bita. Konkretno pridruživanje kodnih riječi simbolima je uobičajeno definirano nekom funkcijom ili tablicom pridruživanja.

Očigledno da broj simbola K koje želimo kodirati određuje koliko minimalno bita n treba imati svaka kodna riječ. S n bita moguće je dobiti $N=2^n$ različitih kombinacija binarnog zapisa pa u općenitom slučaju treba biti zadovoljeno:

$$K \leq N = 2^n \tag{22}$$

Tipično u praksi određeni broj kodnih riječi ostaje neiskorišten jer vrlo često K ne bude točno neka potencija broja 2. To ostavlja mogućnost različitih načina raspoređivanja N kodnih riječi na K elemenata abecede. Pridruživanje za prvi element abecede možemo napraviti na N različitih načina (kodnih riječi). Nakon toga pridruživanje drugom elementu abecede možemo napravit na preostalih N-1 načina itd. U konačnici broj mogućnosti kako možemo napraviti raspodjelu N kodnih riječi između K elemenata abecede je definiran kao:

$$N \cdot (N-1) \cdot (N-2) \ldots (N-K+1) = \frac{N!}{(N-K)!} \tag{23}$$

Kodovi koji služe za prikaz znamenaka zovu se numerički kodovi, a posebice su interesantni oni za kodiranje dekadski znamenaka, tzv. dekadski kodovi.

1.6.1 Dekadski kodovi

Izraz (23) potencijalno daje veliki broj mogućnosti oko načina raspoređivanja kodnih riječi simbolima abecede. To se zorno može vidjeti prilikom kodiranja znamenaka dekadskog

sustava gdje je K=10. Prema uvjetu (22) potrebna su barem n=4 bita u kodnim riječima. Četiri bita omogućavaju $N=2^4$ različitih kombinacija za kodne riječi. Tada sukladno (23), mogući broj različitih kodiranja iznosi:

$$\frac{N!}{(N-K)!} = \frac{16!}{(16-10)!} = 29{,}059 \cdot 10^{29} \tag{24}$$

Postavlja se pitanje da li je iz ovakvo velikog broja mogućih načina kodiranja odabrati ona koja imaju određena povoljna svojstva. U kontekstu dekadskih kodova dva svojstva koja se smatraju povoljnima su aditivnost te samokomplementranost. Prvo navedeno svojstvo podrazumijeva da postoji direktna veza između kodne riječi i prikazane dekadske znamenke. Drugo navedeno svojstvo se odnosi na dvije kodne riječi X i Y, gdje prva kodira neki broj a druga njegov B-1 komplement, te podrazumijeva da se komplementiranjem bitova kodne riječi X (pretvaranje jedinica u nule te obrnuto) dobiva upravo kodna riječ Y. Budući da je dobivanje B-1 komplementa česta i važna operacija u računalnim sustavima samokomplementiranost je svojstvo kome se pridaje velika važnost.

Direktnu povezivost prikazane kodne riječi i dekadske znamenke N moguće je realizirati uporabom jednog od težinskih dekadskih kodova. Kod takvih kodova svaka pozicija bita ima neku svoju težinu, slično kao i kod pozicijskih brojevnih sustava, jedino što te težine nisu nužno u međusobnom odnosu i mogu biti bilo pozitivne, bilo negativne. Pokazuje se da postoji 17 težinskih kodova kod kojih su sve težine pozitivne i 71 težinski *kod* koji imaju neke težine negativne. Težinskim kod moguće je dobiti nazad vrijednost prikazane znamenke izrazom:

$$N = \sum_{i=0}^{n-1} a_i \cdot w_i + D \tag{25}$$

gdje je w_i i-ta težina, n broj bitova kodne riječi, a_i koeficijent (0 ili 1) za i-tu težinu te D konstanta pomaka. Nadalje, može se pokazati da je težinski kod samokomplementirajući ako je zadovoljen uvjet:

$$\sum_{i=0}^{n-1} w_i = 9 \tag{26}$$

1.6.1.1 8421 kod

Vjerojatno najjednostavniji dekadski *kod* je onaj gdje se 4 bitne kodne kombinacije biraju tako da svaka kombinacija odgovara reprezentaciji dekadske znamenke u binarnom brojevnom sustavu. Zbog direktne veze kodnih riječi s reprezentacijom znamenki u binarnom brojevnom sustavu takav *kod* se naziva prirodni binarno kodirani dekadski kod (engl. BCD - Binary Coded Decimal), a skraćeno se izražava kao 8421 *kod* s čime se ističu težine odgovarajućih pozicija u binarnom brojevnom sustavu. BCD kod koristi dakle prvih deset od 16 mogućih kombinacija (Tablica 1.10).

1.6.1.2 2421 (Aikenov kod)

Ovaj dekadski *kod* ima težine 2421. Koristi se prvih pet i zadnjih pet kombinacija od mogućih 16 kombinacija. Zatim *kod* je samokomplementirajući budući da se kodne kombinacije unutar parova B-1 komplementa (0-9, 1-8, 2-7, 3-6, 4-5) se dobivaju jednostavnim komplementiranjem kodne riječi (Tablica 1.10). Npr. za komplementarni par vrijednosti 2-7 imamo korespondentne kodne riječi 0010 te 1101 iz čega očito slijedi da je prijelaz iz jedne u drugu kodnu riječi moguć jednostavnim komplementiranjem bitova. Može se primijetiti da je svaki četverobitni binarni kod samokomplementirajući ako je simetričan ili drugim riječima ako su kodne riječi za komplementarne znamenke jednako udaljene od osi simetrije, koja prolazi sredinom između kombinacija 0111 i 1000.

1.6.1.3 Kod eksces 3

Tablica 1.10 Usporedni prikaz nekoliko dekadskih kodova. U zelenoj boji su prikazane iskorištene kodne kombinacije, a u crvenoj neiskorištene kodne kombinacije.

Kod 8421 (BCD)	
d	BCD
0	**0000**
1	**0001**
2	**0010**
3	**0011**
4	**0100**
5	**0101**
6	**0110**
7	**0111**
8	**1000**
9	**1001**
	1010
	1011
	1100
	1101
	1110
	1111

Kod 2421	
d	2421
0	**0000**
1	**0001**
2	**0010**
3	**0011**
4	**0100**
	0101
	0110
	0111
	1000
	1001
	1010
5	**1011**
6	**1100**
7	**1101**
8	**1110**
9	**1111**

Kod eksces 3	
d	eksces 3
	0000
	0001
	0010
0	**0011**
1	**0100**
2	**0101**
3	**0110**
4	**0111**
5	**1000**
6	**1001**
7	**1010**
8	**1011**
9	**1100**
	1101
	1110
	1111

Kod je dobio ime zahvaljujući činjenici da pridruživanje kodnih riječi nije započelo od kombinacije 0000 kao u slučaju 8421 *koda*, već su preskočene prve tri kodne kombinacije (Tablica 1.10). Eksces 3 *kod* je težinski uz postav vrijednosti D=3 **(25)**. Poznat je još i pod imenom Stibitzov *kod*. Ovaj *kod* je također samokomplementirajući. Značajna prednost mu je i što nema kodne riječi s svim nulama 0000, a što može biti značajno u prijenosu podatka jer se eventualni prekid prijenosa ne može protumačiti kao nula.

1.6.1.4 Bikvinarni kod

Tablica 1.11 Bikvinarni kod. Sastoji se od dvije grupe bitova. U svakoj grupi se nalazi po jedna 1, istaknuta zelenom bojom u tablici.

d	**5**	**0**	**4**	**3**	**2**	**1**	**0**
0	0	**1**	0	0	0	0	**1**
1	0	**1**	0	0	0	**1**	0
2	0	**1**	0	0	**1**	0	0
3	0	**1**	0	**1**	0	0	0
4	0	**1**	**1**	0	0	0	0
5	**1**	0	0	0	0	0	**1**
6	**1**	0	0	0	0	**1**	0
7	**1**	0	0	0	**1**	0	0
8	**1**	0	0	**1**	0	0	0
9	**1**	0	**1**	0	0	0	0

Ovaj *kod* je karakteristična po tome što se sastoji od kodnih riječi s dvije grupe bitova. Prva grupa sadrži dva, a druga pet bitova (Tablica 1.11). U svakoj grupi bitova nalazi se po jedna jedinica odnosno ukupno gledajući dvije jedinice po kodnoj riječi pa će se relativno jednostavno moći detektirati svaka pogreška koja bi promijenila broj jedinica u kodnoj riječi.

Zahvaljujući velikom broju mogućih kombinacija 2^7=128, *kod* ima značajnu zalihost. Bikvinarni kod je težinski sastavljen od težina 504321.

1.6.1.5 Grayev kod

U mnogim primjenama poželjni su kodovi s minimalnom promjenom tj. kodovi gdje se susjedne riječi razlikuju samo u jednom bitu. Takvi kodovi posljedično imaju minimalnu distancu jednaku 1. Jedna od najpopularnijih takvih kodova je Grayev kod. Primjerice, Gray kodovi se koriste u koderima položaja (linearni enkoderi i rotirajući enkoderi). Ako bi se koristio prirodni binarni *kod* tada bi se pri promjeni susjednih položaja trebalo mijenjati istovremeno nekoliko bitova, a što je u praksi o neizvedivo. Kao posljedica će se dogoditi pogrešno očitanje bitova jer se neki bitovi mijenjaju brže, drugi sporije. Dok se ne završi promjena svih bitova, dotične privremene među vrijednosti kodnih riječi dati će značajno različita očitanja, npr. položaja, što može uzrokovati probleme kod upravljanja nekim sustavom. Nasuprot tome, uporabom Grayevog koda mijenja se samo jedan bit te prostorni bliski položaji imati će i kodne riječi koje se razlikuju samo za jedan bit. Grayev kod naziva se i reflektirajući kod zbog jednog od načina njegove konstrukcije, temeljenog na uporabi BCD koda i refleksije /zrcaljenja) toga BCD koda oko osi simetrije. Postupak konstrukcije Grayevog koda zrcaljenjem prikazan je izrazom (27).

$$\begin{array}{cccc} & & & 00 & 000 \\ & & & 01 & 001 \\ & 0 & 00 & 11 & 011 \\ \frac{0}{1} & \frac{1}{1} & \frac{01}{11} & \frac{10}{10} & \frac{010}{110} \\ & 0 & 10 & 11 & 111 \\ & & & 01 & 101 \\ & & & 00 & 100 \end{array} \tag{27}$$

U prvom stupca izraza (27) su na početku napisana dva bita 0 i 1. Zatim su u drugom stupcu oba bita napisan iznad razlomačke crte (crte zrcaljenja), a ispod dotične crte je napisan gornji raspored bitova u zrcalnom odnosu, tj. prvo 1 pa 0. Nakon toga je trećem stupcu gornjim kombinacijama bitovima nadodana 0 slijeva, a donjim kombinacijama bitova 1. U četvrtom stupcu su sve četiri kombinacije riječi iz trećeg stupca napisane iznad razlomačke crte, a

ispod razlomačke crte su napisane kombinacije riječi koje su u zrcalnom odnosu s gornjim kombinacijama.

Tablica 1.12 Grayev kod.

			Binarni Grayev kod
		0000	
		0001	
		0011	
Dekadski Grayev kod	0	**0010**	
	1	**0110**	
	2	**0111**	
	3	**0101**	
	4	**0110**	
	5	**1000**	
	6	**1001**	
	7	**1011**	
	8	**1010**	
	9	**1110**	
		1111	
		1101	
		1110	

Pa je ponovno u petom stupcu prikazano kako se gornjim kombinacijama riječi nadodaje 0 s lijeva, a donjim kombinacijama se nadodaje 1 s lijeva. U ovome času dobiva se trobitni Grayev kod, a postupak je moguće nastaviti i dalje u cilju konstrukcije Grayevog koda s još više bitova. U kontekstu kodiranja dekadskih znamenki govori se o dekadskom Grayevom kodu koji je u odnosu na binarni Grayev *kod* pomaknut za tri kodne riječi, tzv. eksces 3 Grayev *kod* (Tablica 1.12). Razlog tome je da se dobije slučaj gdje se kodne kombinacije za znamenke 0 i 9 također razlikuju samo u jednom bitu, a što također bitno u mnogim primjenama.

1.7 Otkrivanje i ispravljanje pogrešaka u nizovima bitova

Prikaz praktički svih informacija u računalu (digitalnom sustavu) pomoću zapisa bitova, tj. niza sastavljenog od jedinica i nula, je u praksi podložan utjecaju smetnji te zbog toga i mogućim pogreškama u zapisu. Razlikujem jednostruku pogrešku kada samo na jednom od bitova 0 prijeđe u 1 ili obrnuto, te višestruku pogrešku kada na više od jednog bita dođe do neželjene zamijene 0 u 1 ili 1 u 0. Novonastala kodna riječ može biti jedna od neispravnih kodnih riječi pa je u tome slučaju detekcija i ispravljanje značajno olakšano u usporedbi ako novonastala riječ postane jedna od ispravnih kodnih riječi. U potonjem slučaju je bez dodatnih saznanja uz npr. širi kontekst informacije praktički nemoguće otkriti da je uopće došlo do pogreške.

1.7.1 Zaštita pomoću zalihosti kodni riječi

Jedan od najjednostavnijih principa za otkrivanje i eventualno ispravljanje pogrešaka koristi se zalihosti koda tj. činjenicom da za neku primjenu nisu iskorištene sve teoretski raspoložive kombinacije za kodne riječi već neki manji podskup. Zahvaljujući toj činjenici moguće je otkriti da je došlo do jednostruke ili višestruke pogreške ukoliko se neka ispravna kodna riječ pretvori u jednu od neispravnih (neiskorištenih) kodnih riječi. U pravilu što je veća zalihost koda (veći broj neiskorištenih kodnih riječi) to će stupanj zaštite biti veći. Konkretan broj (višestrukih) pogreška koje je moguće detektirati određujemo analizirajući razliku između kodnih riječi. Zato nam može poslužiti Hammingova (R. W. Hamming, 1915-1998) distance (udaljenost) između dviju kodnih riječi. Hammingova udaljenost je broj bitova u kojima se

dvije kodne riječi razlikuju. Drugim riječima predstavlja broj bitova potreban da se jedna kodna riječ pretvori u drugu kodnu riječ. Formalno gledajući Hammingova udaljenost *d* između dviju kodnih riječi *x* i *y* se može definirati izrazom:

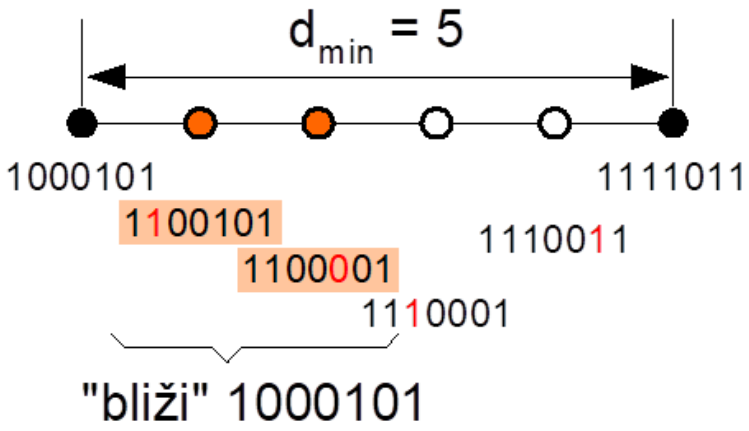
$$d = |x \oplus y| \tag{28}$$

Operator ⊕ u gornjem izrazu predstavlja tzv. XOR operaciju nad binarnim brojevima koja za sumu dvije binarne znamenke daje jedan samo ako su one različite, a inače je suma nula. Također XOR u gornjem izraz primijenjen je po korespondentnim bitovima riječi x i y. Konačno operator |·| u izrazu (28) predstavlja broj jedinica u nekoj kodnoj riječi. Najveći broj pogrešaka *t* koje je moguće otkriti definiran je minimalnom Hammingovom distancom d_{min} koja određuje najmanji broj bitova potreban da se iz jedne ispravne kodne riječi prijeđe u neku drugu ispravnu kodnu riječ, istoga koda. Očigledno da mora vrijediti:

$$d_{min} > t + 1 \tag{29}$$

Prema tome za različite kodove nam je interesantna upravo minimalna Hammingova distance gdje svi oni kodovi koji nemaju $d_{min}>1$ ne mogu koristit gornji princip zalihosti za otkrivanje pogrešaka (Tablica 1.13). Pored otkrivanja moguće je provesti i ispravljanje pogrešaka ukoliko se uvede pretpostavka da je najbliža ispravna kodna riječ, u odnosu na dotičnu neispravnu kodnu riječ, ona prava - ispravna kodna riječ. Slika 1.11 daje primjer dviju ispravnih kodni riječi 1000101 i 1111011 uz minimalnu distancu $d_{min}=5$: Ukoliko se pojavila neka od neispravnih kodnih riječi 1100101 ili 1100001 pretpostavit će se da je ispravna kodna riječ 1000101 budući da je za spomenute dvije neispravne kodne riječi najbliža ispravna kodna riječ upravo 1000101. Da bi za neki kod mogli primijeniti ne samo detekciju već i ispravljanje *t* pogrešaka na spomenutom principu, postavljaju se još oštriji zahtjevi na iznos minimalne distance koda d_{min}. Mora biti zadovoljeno:

$$d_{min} > 2 \cdot t + 1 \tag{30}$$



Slika 1.11 Princip odabira najbliže kodne riječi za ispravnu kodnu riječ.

Tablica 1.13 Primjeri minimalne Hammingove udaljenosti d_{min}

Kod	dmin
8421 kod	1
bikvinarni kod	2
Grayev *kod*	1

1.7.2 Zaštita paritetom

Očigledno da za kodove čiji minimalne distance d_{min} ne zadovoljavaju uvjete (28) i/ili (29) nije moguće primijeniti niti ispravljanje niti otkrivanje pogreška. Za takve kodove, kao naravno i za one koji zadovoljavaju uvjete (28) i/ili (29), moguće je primijeniti drugi alternativni način zaštite proširivanjem originalne kodne riječi s paritetnim bitom. Vrijednost paritetnog bita ovisi o broju jedinica originalne kodne riječi i tome želi li se primijeniti neparni ili parni paritet. Ukoliko se primjenjuje neparni paritet, a originalna kodna riječ već ima neparni broj jedinica tada će vrijednost paritetnog bita biti postavljena na nulu. U suprotnome ako je broj jedinica bio paran tada paritetni bit poprima vrijednost jedan. Znači ukupni broj jedinica u cjelokupnoj riječi treba biti neparan. Slično tome, primjeni li se parni

paritet, a originalna riječ već ima parni broj jedinica tada se paritetni bit postavlja na nulu, tj. na jedinicu ako je originalna riječ imala neparni broj jedinica. Dakle, u ovome slučaju ukupna riječ treba imati parni broj jedinica. Tablica 1.14 prikazuje primjere kodiranja paritetnim bitom kod odabira bilo parnog, bilo neparnog pariteta i za nekoliko znakova čiji je originalni kod određen kodom ASCII (engl. American Standard Code for Information Interchange). Do pogreške može doći i na samome zaštitnom bitu, ali ovakvim načinom zaštite moguće je utvrditi samo da je došlo do pogreške na nekom od bitova ukupne kodne riječi, a što uključuje i zaštiti bit.

Tablica 1.14 Primjer zašite kodiranja paritetnim bitom. Masno otisnuti i podcrtani (najlijeviji) bit predstavlja vrijednost paritetnog bita

Znak		Parni paritet	Neparni paritet
V	1010110	**0**1010110	**1**1010110
B	1000010	**0**1000010	**1**1000010
L	1001100	**1**1001100	**0**1001100
G	1000111	**0**1000111	**1**1000111
T	1010100	**1**1010100	**0**1010100

Provjerom vrijednosti jednog paritetnog bita moguće je detektirati pojavu jednostruke (neparne) pogreške. Pojavu dvostruke pogreške, odnosno parnih pogrešaka nije moguće detektirati na gornji način. Međutim u praksi je najčešća pogreška jednostruke pogreške pa je i inače velik broj zaštitnih kodova namijenjen upravo detekciji i eventualno ispravljanju jednostrukih pogrešaka. Bez mogućnosti ispravljanja pogreške nužno je ponoviti operaciju prijenosa podataka. Zato se prilikom kodiranja jednim paritetnim bitom postavlja se pitanje da li je moguće povećanjem broja paritetnih bitova povećati stupanj zaštite te eventualno omogućiti i ispravljanje jednostruke pogreške, a ne samo detekciju njene pojave. Dvodimenzijski kod je primjer proširenja zaštite s jednog na više paritetnih bitova, s

mogućnošću ispravljanja pogreške. Ideja je konstruirati 2D matricu informacijskih bitova te na kraju (početku) svakog retka i na dnu (vrhu) svakog stupaca matrice dodati po jedan zaštitni paritetni bit. Na taj način kodira se cijeli blok kodnih riječi i omogućava ispravljanje jednostruke pogreške. Tablica 1.15 pokazuje kako se jedan cijeli blok riječi može skupno zaštititi s čitavim nizom zaštitnih bitova, po recima i stupcima.

Tablica 1.15 Primjer zaštite kodiranja dvodimenzijskim kodom uz uporabu uzdužnih i poprečnih zaštitnih paritetnih bitova (parni paritet uporabljen).

Znak	Originalna kodna riječ							Paritetni bit
V	1	0	1	0	1	1	0	**0**
B	1	0	0	0	0	1	0	**0**
L	1	0	0	1	1	0	0	**1**
G	1	0	0	0	1	1	1	**0**
T	1	0	1	0	1	0	0	**1**
paritetni bit	**1**	**0**	**0**	**1**	**0**	**1**	**1**	

Tablica 1.16 Primjer nalaženja pogreške prilikom zaštite kodiranja dvodimenzijskim kodom uz uporabu uzdužnih i poprečnih zaštitnih paritetnih bitova (parni paritet uporabljen).

Znak	Originalna kodna riječ							Paritetni bit
V	1	0	1	0	1	1	0	**0**
B	1	0	0	0	0	1	0	**0**
L	1	0	0	1	1	0	0	**1**
G	1	0	***1***	0	1	1	1	**0**
T	1	0	1	0	1	0	0	**1**
paritetni bit	**1**	**0**	**0**	**1**	**0**	**1**	**1**	

Pojava jednostruke pogreške unutar bitova originalne kodne riječi izazvat će neslaganje s vrijednostima paritetnog bita u korespondentnom retku i korespondentnom stupcu. Npr. neka se je treći bit znaka 'G' uslijed pogreške promijenio iz 0 u 1 (

Tablica 1.16). Promatrajući samo jedan od dotičnih zaštitnih bitova ne možemo točno odrediti na kojem je bitu došlo do pogreške. Ali zato sjecište horizontale retka paritetnog bita s vertikalom stupca drugog zaštitnog bita označava mjesto pogreške bita.

Uvođenje zaštitnih bitova povećava ukupan broj bitova kodne riječi. Neka je n ukupni broj bitova kodne riječi, k broj originalnih informacijskih bitova, a $r=n-k$ broj zaštitnih bitova. Za kvantitativan opis zalihosti (snage zaštite) definira se faktor zalihosti R:

$$R = \frac{r}{n} \tag{31}$$

1.7.3 Hammingovi kodovi

Primjer dvodimenzijskog koda pokazao je na jednostavan način kako se uvođenjem dodatnih paritetnih bitova može dobiti relativno jednostavan mehanizma detekcije i ispravljanja pogreške. Ipak broj dodatnih bitova je proporcionalan veličini informacijskog bloka kojeg se želi zaštititi te u tome smislu taj broj nije zanemariv. Postoji i efikasniji način korištenja zaštitnih bitova, a jedan od najpoznatijih je formiran kroz princip Hamingovog kodiranja. Ono se također temelji na ideji da se jedan informacijski bit štiti s više od jednim zaštitnim paritetnim bitom, ali to čini tako da se pojedine manje grupe bitova štite s određenim zaštitnim bitovima (višestruko ispitivanje pariteta) i u konačnici to čini na puno efikasniji način u smislu potrebnog broja zaštitnih bitova za neku količinu informacijskih bitova. Vjerojatno najpoznatija inačica Hammingovog zaštitnog koda je ona koja se odnosi na detekciju i ispravljanje jednostruke pogreške. Tablica 1.17 ilustrira način funkcioniranja takvog zaštitnog kodiranja. U dotičnom primjeru ukupna kodna riječ sastavljena je od tri zaštitna bita $C_2C_1C_0$ i četiri informacijska bita $k_3k_2k_1k_0$. Zaštitni bitovi se postavljaju na pozicije određene potencijom broja 2, tj. na pozicije 2^0, 2^1, 2^2 itd., a informacijski bitovi na ostale pozicije. Svaki zaštitnih bitova, preko pariteta, štiti druge pozicije u ukupnoj kodnoj riječi. U odgovarajućim bojama je prikazano koje bitove štiti pojedini zaštitni bit. Tako prvi zaštitni C_0 bit štiti svoju poziciju zatim 3, 5, 7 i tako dalje svaku neparnu poziciju do kraja

riječi. Drugi zaštitni bit je zadužen štititi svoju drugu poziciju te odmah sljedeću udesno, zatim dvije pozicije preskače pa štiti sljedeće dvije pozicije i tako dalje bi se nastavilo po istom uzroku do kraja kodne riječi. Konačno posljednji zaštitni bit C_2 uz svoju poziciju štiti sljedeće tri pozicije udesno, pa bi nakon toga preskočio štićenje pozicija 8, 9, 10, 11 (nije prikazano u tablici) te nastavio štititi pozicije 12, 13, 14 i 15 itd. Važno je uočiti da se područja štićenja od strane pojedinih zaštitnih bitova preklapaju. Zahvaljujući takvom mehanizmu paritetnog štićenja bitova, pronalazak eventualne pogreške na nekom bitu će usmjeren kroz doprinose vrijednosti svih zaštitnih bitova. Primjerice, pretpostavimo da je usporedba očekivane vrijednosti pariteta za C_2 različita od primljene vrijednosti za zaštitni bit C_2 , čime nam C_2 signalizira pojavu pogreške. Područje koje štiti C_2 su pozicije u kodnoj riječi 5, 6 i 7 (Tablica 1.17) te promatrajući C_2 sam po sebi ne znamo na kojem je od ta četiri bita došlo do pogreške. Zato nadalje analiziramo vrijednost zaštitnog bita C_1 te i za njega uspoređujemo očekivanu vrijednosti pariteta s primljenom vrijednosti za C_1. Pretpostavimo da je došlo i tu do razlike što bi nam signaliziralo da je do pogreške došlo na nekom od bitova 6 ili 7, s obzirom na područje odgovornosti štićenja od strane bita C_1. Konačni zaključak da li je pogreška nastala na bitu 6 ili 7 bit donosimo analizom vrijednosti za zaštitni bit C_0. Pretpostavimo da za taj bit nema razlike između očekivane njegove vrijednosti i konkretno primljene, tj. da što se njega tiče nije nastala nikakava (jednostruka) pogreška. Prisjetimo se da C_0 štiti svaku neparnu poziciju u kodnoj riječi, pa tako i poziciju 7. Budući da za njegov skup štićenih bitova nema pogreške te na osnovu prethodnih zaključaka glede bitova C_1 i C_2, konačno zaključujemo da je pogreška nastala na poziciji bita 6 te da je on taj kojem treba promijeniti vrijednost da bi se dobila očekivana točna kodna riječ.

Tablica 1.17 Hammingov kod: oznake štićenja pojedinih bitovnih mjesta u kodnoj riječi pomoću tri zaštitna bita $C_2C_1C_0$ i u slučaju četiri informacijska bita $k_3k_2k_1k_0$. Jedinice tablici, prikazane u korespondentnim bojama s oznakama zaštitnih bitova, pokazuju na pozicije unutra ukupne kodne riječi koje štite pojedini zaštitni bitovi.

Zaštitni bitovi	Pozicija bita u kodnoj riječi						
	1	2	3	4	5	6	7
C_2	0	0	0	1	1	1	1
C_1	0	1	1	0	0	1	1
C_0	1	0	1	0	1	0	1
	C_0	C_1	k_0	C_2	k_1	k_2	k_3

U praksi se detekcija nastajanje pogreške i mjesta na kojem je pogreška nastala provodi izračunom tzv. sindroma *S*: uzorak bitova koji ukazuju na mjesto pogreške. Kao što je već rečeno, promatra se primljena kodna riječ (npr. pristigla određenim komunikacijskim kanalom) tako što se prvo izračunaju vrijednosti pariteta temeljem pristiglih informacijskih bitova. Konkretno za primjer od tri zaštitna $C_2C_1C_0$ bita i četiri informacijska bita $k_3k_2k_1k_0$ (Tablica 1.17) te u slučaju parnog pariteta, iznosi za zaštitne bitove bi se nalazili preko izraza:

$$\begin{aligned} C_0 &= k_0 \oplus k_1 \oplus k_3 \\ C_1 &= k_0 \oplus k_2 \oplus k_3 \\ C_2 &= k_1 \oplus k_2 \oplus k_3 \end{aligned} \tag{32}$$

Odnosno ako se radilo o nepranom paritetu tada bi se vrijednosti za zaštitna $C_2C_1C_0$ bita nalazile kao:

$$\begin{aligned} C_0 &= k_0 \oplus k_1 \oplus k_3 \oplus 1 \\ C_1 &= k_0 \oplus k_2 \oplus k_3 \oplus 1 \\ C_2 &= k_1 \oplus k_2 \oplus k_3 \oplus 1 \end{aligned} \tag{33}$$

Nadalje se promatra uzorak zaštitnih bitova $C'_2C'_1C'_0$ koji su pristigli s primljenom kodnom riječi tako da se usporedi s prethodno izračunatim $C_2C_1C_0$ vrijednostima preko operacije XOR (modulo 2 suma):

$$S = C_0C_1C_2 \oplus C'_0C'_1C'_2 \tag{34}$$

gdje je S sindrom koji ima vrijednost nula u slučaju da nije detektirana pogreška, tj. za vrijednost različitu od nule sindrom ukazuje na poziciju unutar kodne riječi gdje je došlo do pogreške (uključujući i pozicije na samim zaštitnim bitovima). Izrazi (32) i (33) također mogu poslužiti prilikom konstruiranja ukupne štićene riječi za četiri informacijska bita. Četiri informacijska bita $k_3k_2k_1k_0$ mogu imati jedna od 16 mogućih uzoraka bitova. Tablica 1.18 pokazuje za svaki od 16 slučaja bitovnih uzoraka kako bi izgledala ukupna kodna riječ štićena Hammingovoim kodom, uz pretpostavku parnog pariteta.

Tablica 1.18 Izgled zaštićenih četverobitnih uzoraka bitova $k_3k_2k_1k_0$ s Hammingovim kodom, uz parni paritet.

Izvorna kodna riječ bez zaštite				Zaštićena kodna riječ						
k_3	**k_2**	**k_1**	**k_0**	**C_0**	**C_1**	**k_0**	**C_2**	**k_1**	**k_2**	**k_3**
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	0	0	0	0
0	0	1	0	1	0	0	1	1	0	0
0	0	1	1	0	1	1	1	1	0	0
0	1	0	0	0	1	0	1	0	1	0
0	1	0	1	1	0	1	1	0	1	0
0	1	1	0	1	1	0	0	1	1	0
0	1	1	1	0	0	1	0	1	1	0
1	0	0	0	1	1	0	1	0	0	1
1	0	0	1	0	0	1	1	0	0	1
1	0	1	0	0	1	0	0	1	0	1
1	0	1	1	1	0	1	0	1	0	1
1	1	0	0	1	0	0	0	0	1	1
1	1	0	1	0	1	1	0	0	1	1
1	1	1	0	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1

Uvođenjem *r* zaštitnih bitova za originalno *k* informacijskih bitova neke riječi, ukupna kodna riječ se proširuje na *n*=*k*+*r* bitova. Da bi se Hammingovim kodom (sindromom) za detekciju i ispravljanje pogreške korištenjem *r* zaštitnih bitova moglo štititi *n* pozicija ukupne kodne riječi treba biti zadovoljeno:

$$2^r \geq k + r + 1 \tag{35}$$

Gornji izraz uključuje i slučaj kada sindrom od *r* bita ima vrijednost nula te s time ukazuje da nije došlo do pogreške. Značajna prednost Hammingovog je efikasnost gdje se uz relativno malu zalihost (31), s obzirom na broj dodatnih zaštitnih bitova *r* potrebnih za zaštitu početnih *k* informacijskih bitova, postiže visoki stupanj zaštite. Štoviše, efikasnost Hammingovog koda raste s povećanjem broja informacijskih bitova *k* gdje udio zaštitnih bitova *r* u ukupnoj broju bitova *n* postaje sve manji i manji kako broj *k* raste.

Tablica 1.19 Odnos zaštitnih bitova *r* i informacijskih bitova *k*, za Hammingov kod, definiran izrazom $2^r \geq k+r+1$.

Broj informacijskih bitova *k*	Broj zaštitnih bitova *r*	Ukupna duljina kodne riječi *n*
1	2	3
4	3	7
11	4	5
26	5	31
57	6	63
120	7	127

2 Računala i programiranje

2.1 Računala

Današnji svijet nije moguće zamisliti bez računala. Okruženi smo različitim tipovima, od onih klasičnih stolnih i prijenosnika, velikih poslužitelja u kompanijama do pametnih telefona i malih kontrolera koji su ugrađuje u sve više uređaja. Bez obzira na njihovu veličinu i procesnu moć, svi se temelje na istom principu i imaju zajedničku povijest. Prva ne-mehanička računala su se pojavila u prvoj polovici 20 st., temeljila su se na elektronici (tada elektronske cijevi) i njihova namjena je bila računanje složenih matematičkih operacija. Elektroničke komponente možemo promatrati kao niz sklopki koje prepoznaju dva stanja. Stanje kada ima ili nema struje koje označavamo s 1 i 0. S toga i računanje koje je moglo biti obavljano njima nije moglo biti računanje kakvo mi koristimo u matematici s bazom 10, nego je bilo računanje s bazom 2 – **binarni sustav**. Stoga kada govorimo o računanju na računalu temeljna jedinica informacije nosi jedno od ta dva stanja, a tu temeljnu jedinicu nazivamo **bit**. Važno je primijetiti da za razliku od dekadskog sustava gdje s jednom znamenkom možemo prikazati 10 brojeva (od 0 do 9) u binarnom sustavu možemo prikazati samo dva (0 i 1) i prije računanja moramo pretvoriti broj iz dekadskog u binarni. Na primjer broj 8, koji je u dekadskom sustavu $8 * 10^0$ u binarnom prikazujemo kao 1000 odnosno $\mathbf{1}*2^3 + \mathbf{0}*2^2 + \mathbf{0}*2^1 + \mathbf{0}*2^0$ i za njegovu pohranu nam je potrebno 4 bita. Važno je napomenuti da je računanje s bitovima posljedica građe naših elektroničkih računala i da recimo kvantno računalo koje je u razvoju vrijede sasvim drugačija pravila gdje osnovnu jedinica informacije nazivamo **qubit** i u koju je moguće pohraniti više informacije nego u **bit**. Koristeći pune mogućnosti koje tehnologija pruža, kvantna računala će omogućiti puno brže računanje.

Temeljne operacije koje možemo raditi s elektroničkim računalom se svode na zbrajanje, logičke operacije (I, ILI i NE) te korištenje posmaka. Operacije se izvode s jednim ili dva operanda. Primjer posmaka je kada radimo posmak u lijevo za jedno mjesto broja 4, binarno 100. Pošto se pomičemo u lijevo dodajemo na zadnje mjesto broj 0 i dobijemo broj 1000 što odgovara 8 u dekadskom sustavu. Da smo napravili pomak u desno, onda bi izgubili zadnju 0 i dobili 2. Važno je primijetiti da jedan pomak u lijevo je zapravo operacija dijeljena s 2 a jedan pomak u desno operacija množenja. Za množenje i dijeljenje sa 4 bi trebalo napraviti

posmak 2 bita u lijevo odnosno desno. Na ovaj način umjesto uzastopnog zbrajanja množenje s brojevima koji su potencije broja 2 je jako ubrzano. Sve druge operacije se svode na osnovne operacije.

Da bi smo mogli raditi operacije s računalom potrebno je da obadva operanda su iste unaprijed definirane duljine. Pretpostavimo da je duljina svakoga operanda 8 bitova. U slučaju kada nam treba manje bitova za prikaz broja onda na sva ostala mjesta ostavimo 0. Tako ćemo broj 8 prikazati kao 00001000. Važno je primijetiti da smo na ovaj način ograničeni koliko brojeva možemo prikazati. Duljina operanda s kojim računalo radi je obično višekratnik broja 8. Zapis od 8 bitova se naziva **Byte** odnosno oktet na hrvatskom. Broj bitova s koje obrađuje računalo s naziva **riječ** (*engl.* word). Današnja računala obavljaju operacije obično s 32 odnosno 64 bita i zato ih nazivamo 32-bitna i 64-bitna računala.

Osim računski operacija podatke moramo omogućiti i njihovu pohranu. Ukoliko želimo zbrojiti tri broja prvo moramo zbrojiti dva broja i nakon toga rezultat te operacije moramo negdje pohraniti da bi smo ga mogli koristiti za zbrajanje s trećim brojem. Pohranjujemo ih u privremenu ili trajnu memoriju. Privremenu memorija koja je aktivna dok je računalo upaljeno nazivamo RAM (*engl.* random-access memory), a trajna memorije je obično disk. Danas obično koristimo dva tipa diskova HDD (*engl.* hard disk drive) i SSD (*engl.* solid state drive) s time da je čitanje i pisanje na SDD disk puno brže, ali isto tako SSD imaju kraći vijek trajanja.

Važan dio građe računala su ulazne i izlazne jedinice preko kojih unosimo podatke odnosno ih iznosimo iz računala. Primjeri ulaznih jedinica su tipkovnica, miš, zaslon osjetljiv na dodir, a izlaznih su: zaslon i pisač.

Kada povežemo dio za računanje (obično ga nazivamo procesorom), jedinice za pohranu podataka (memorija) i ulazno/izlazne jedinice dobijemo računalo. Obično dijelove računala zajednički nazivamo sklopovlje odnosno *hardware*.

2.2 Programska podrška

Samo računalo ne može funkcionirati samo od sebe, potrebno je mu je dati instrukcije. Stoga je potrebno osmisliti način na koji će računalo obrađivati i spremati podatke. Taj dio

nazivamo programskom podrškom odnosno softver. Softver ugrubo možemo podijeliti na sistemski i korisnički. Sistemski softver vodi računa o funkcioniranju našeg računala, načinu na koji se podaci prenose između njegovih dijelova te pohranjuju. Korisnički softver su programi koje svakodnevno koristimo poput tekstualnih i tabličnih procesora (npr. Microsoft Word i Excel), pretraživača Interneta, igara i slično. Obično se za korisnički softver koristi i naziv aplikacija.

Softver se sastoji od niza instrukcija koje se izvršavaju na procesoru računala. Ne smijemo zaboraviti da instrukcije koje se izvršava na procesoru se sastoje od niza nula i jedinica pri čemu se obavljaju jednostavne matematičke ili logičke operacije. Kako bi pisanje takvog softvera oduzelo puno vremena i zahtjeva dobro poznavanje arhitekture pojedinog procesora s vremenom su se razvili viši programski jezici u kojima se pišu čovjeku razumljive naredbe koje se kasnije prevode u strojne instrukcije koje računalo razumije. Svaki od viših programskih jezika ima već napisan velik broj funkcija koje se spremaju u biblioteke koje se onda mogu koristiti pri izradi novoga koda. Primjer takvih funkcija su funkcije za pisanja na ekran ili unošenje teksta preko tipkovnice, složenije matematičke operacije i slično.

Softver možemo izvoditi na različitim tipovima uređaja pa ga stoga dijelimo na: desktop aplikacije, mobilne aplikacije, softver koji se izvršava na poslužiteljima, ugradbeni softver. Softver može biti instaliran na računalo ali se može pokretati udaljeno gdje korisnik pokreće aplikacije koje se nalaze na nekom poslužitelju. U tome slučaju razlikujemo dva softvera: (i) klijentski softver – manji dio koji se nalazi na računalu korisnika koji služi za komunikaciju i prikaz rezultata i (ii) softver na poslužitelju na kome se izvršava glavnina koda i šalju rezultati klijentu. Tipičan primjer je pretraživanje interneta gdje se web klijent nalazi u pretraživaču korisnika, a sadržaj na udaljenom web poslužitelju.

Važan aspekt suvremenih računala predstavlja sigurnost korisnika. Stoga bi i virusni softver mogli staviti u posebnu kategoriju softvera čiji je cilj otimanje podataka od strane korisnika, namjerno rušenje operacijskog sustava kao i neovlašteno korištenje njegovih resursa.

Prema dozvoli korištenja softver možemo podijeliti u tri kategorije: komercijalni softver za koga je potrebno imati plaćenu licencu, softver koji se može besplatno koristiti (*engl.*

freeware) i softver otvorenoga koda pri čemu se može besplatno po potrebi i prilagođavati sam kod.

2.2.1 Operacijski sustav

Kao što živi organizmi na početku života imaju već određene funkcije poput mogućnosti kretanja, hranjenja isto tako za računalo potrebno mu je omogućiti da može funkcionirati. Stoga je prva zadaća sistemskog *softwarea* omogućiti rad svih gradivnih dijelova računala i njihovu međusobnu komunikaciju. Sve te instrukcije se sastoje od nula i jedinica i to je jedini jezik koji računalo razumije i stoga je potrebno sve podatke i instrukcije pretvoriti u taj dio. Računalni stručnjaci koji rade na razvoju sistemskog *softwarea* moraju biti jako dobro upoznati s građom računala. Kada su nastala prva računala znanstvenici koji su radili sa njima su morali znati sve te detalje. Kako su se računala razvijala tako je došlo i do razvoja tih osnovnih funkcionalnosti koje se odvojene od korisničkih programa. Današnja računala funkcioniraju tako da pri njihovom uključivanju su spremna za pokretanje korisničkih programa. To nam omogućava sistemski *software* koji nazivamo operacijski sustav koji se brine o funkcioniranju pojedinih komponenti. Stoga kada dodajemo novu komponentu u naše računalo moramo dodati i programsku podršku koja se modularno uključi u operacijski sustav. Ta programska podrška (*engl.* driver) za pojedine komponente sklopova omogućava njihov ispravan rad i komunikaciju s ostalim dijelovima računala. Osim drivera za funkcioniranje pojedinih dijelova računala važan je tzv. *firmware* – sistemski *software* koji se dolazi zajedno s pripadnim sklopovljem i omogućava njegovu kontrolu na niskoj razini. *Firmware* se obično pohranjuje na memoriji koja se nakon gašenja računala ne gubi, no može se posebnim postupkom promijeniti. Te memorije se skupno nazivaju ROM (engl. Read only memory) i postoje razni tipovi kao što su recimo EPROM (*engl.* erasable programmable read-only memory) ili EEPROM (*engl.* electrically erasable programmable read-only memory). Specijalni tip EEPROM memorije su tzv. *flash* memorije koje danas možemo naći u različitim elektroničkim uređajima kao i popularnim USB memorijama. Jedan od najvažnijih *firmwarea* u računalu je BIOS (*engl.* Basic Input/Output System) koji se koristi za inicijalizaciju sklopovlja prilikom paljenja računala i pruža usluge u realnom vremenu operacijskom sustavu i programima.

Danas dominiraju dva tipa operacijskih sustava: temeljeni na Unix i Microsoft Windows platformama. Operacijski sustavi temeljeni na Unix platformi su npr. Linux, Android, BSD, iOS, MacOS. Primjer operacijskih sustava temeljenih na Microsoft Windows platformi su MS Windows 10 i Windowa 10 Mobile.

Za uspješan rad računala važan je način na koji spremamo podatke na medije za trajnu pohranu. Podatke koje spremamo nazivamo datotekama a metode i strukture podataka koje operacijski sustav koristi za njihovu pohranu nazivamo datotečni sustav.

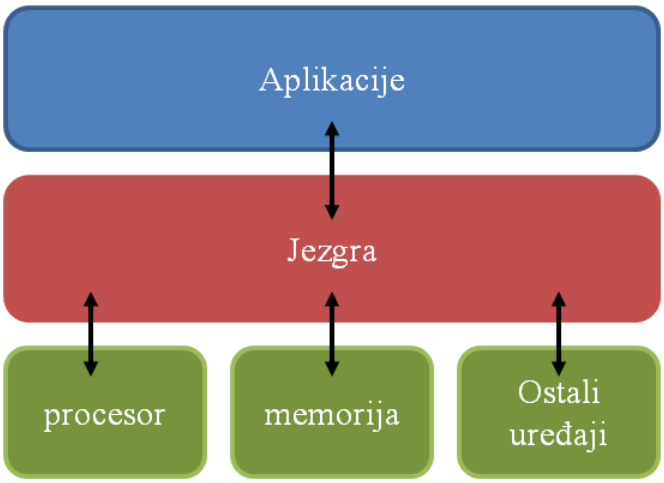
Suvremeni operacijski sustavi na računalima i mobilnim telefonima trebaju omogućavati neke od ovih funkcionalnosti: rad u realnom vremenu (engl. real time), podržavati jednog ili više korisnika (engl. multiuser) kao i istovremeno izvršavanje više aplikacija (engl. multitasking). Zavisno o tim funkcionalnostima možemo vrednovati pojedine operacijske sustave. Posebni operacijski sustavi su oni koji omogućavaju rad na grupi računala (engl. distributed) i na ugradbenim računalnim sustavima (engl. embedded). Vrlo često kada želimo isprobati kako se naša aplikacija ponaša na različitim operacijskim sustavima ili želimo isprobati korištenje različitih operacijskih sustava koristimo tzv. virtualne operacijske sustave gdje unutar jednog operacijskog sustava podignemo drugi - virtualni i nastavimo u njemu raditi kao da smo ga podignuli na računalu.

Komponente operacijskog sustava su:

- Jezgra
- Mrežno povezivanje
- Sigurnost
- Korisničko sučelje

Jezgra operacijskog sustava

Uloga jezgre operacijskog sustava je kontrola sklopovlja (zajedno s *driverima* i *firmwareom*). Jezgra upravlja privremenom RAM memorijom i određuje pristup programima pojedinim dijelovima sklopovlja, organizira podatke za spremanje na datotečne sustave.



Slika 2.1 Jezgra operacijskog sustava povezuje aplikacije sa sklopovljem računala

Jezgra operacijskog sustava omogućuje između aplikacije i sklopovlja na način da se kreira proces koji dodjeljuje svakoj instanci aplikacije memorijski prostor i ostale resurse, učitava aplikaciju u memoriju i pokreće ju. Nakon pokretanja aplikacija međudjeluje s korisnikom i sklopovljem.

Važan koncept operacijskih sustava su prekidi koji pružaju efikasan način funkcioniranja računala. Prekidi mogu doći od programa koji se trenutno izvršavaju ili pojedinih uređaja sklopovlja. Kada dođe zahtjev za prekidom, jezgra odlučuje kako će reagirati na njega ovisno o visini njegovog prioriteta. Obično se prekida izvršavanje trenutnog programa, sprema mjesto gdje je programa prekinut da bi se moglo nastaviti izvršavanje kasnije te izvrši programski kod vezan uz prekid.

Suvremeni procesori imaju nekoliko modova rada, minimalno dva: korisnički i nadzorni mod. Običajno nadzorni mod omogućava nadzor nad svim resursima dok je korisnički mod obično ograničen.

Upravljanje memorijom je ključno za uspješno izvođenje operacija. Neophodno je da svaki program i svako instanca pojedinog programa imaju svoj dio memorije kojem drugi nemaju pristup jer se u suprotnom može dogoditi međusobno brisanje podataka ili instrukcija.

Bitan koncept kojeg podržavaju svi suvremeni operacijski sustavi je virtualna memorija. Koristeći ovu tehniku korisnik ima dojam da ima više memorije nego što je to stvarno.

Korisnik „vidi“ samo virtualne adrese dok operacijski sustav te adrese preslikava u stvarne adrese u fizičkoj memoriji. Iako korisniku je dodijeljen neprekinut segment virtualne memorije, on se u stvarnosti može sastojati od više međusobno odijeljenih segmenata fizičke memorije. Prilikom dodjeljivanja memorije, operacijskih sustav često dodijeli blokove memorije koji se u stvarnosti nalaze na nekom mediju za trajnu pohranu poput diska. Na taj način ukupna dostupna memorija je veća od fizičke veličine RAM memorije. Navedeni blokovi se nazivaju *pages* i shodno tome ovaj način se naziva *pagging*. Operacijski sustav predviđa koji dijelovi memorije neće trebati uskoro pa ih pohranjuje na disk. U slučaju kada operacijski sustav želi pristupiti *page* koja trenutno nije u RAM memoriji, operacijski sustav oslobađa dio memorije u RAM-u i u njega učitava podatke s traženog *pagea* koji je na disku.

Kada radimo na računalu rijetko imamo pokrenutu samo jednu aplikaciju nego je to obično više njih. Npr. pišemo tekst i slušamo glazbu. Taj koncept nazivamo multitasking, no procesor u stvarnosti obavlja u jednom vremenskom trenutku jednu instrukciju jedne aplikacija. Multitasking se temelji na tome da svaki proces uzme dio vremena, zaustavi se dok se odradi dio drugog procesa i nastavi rad kada po rasporedu opet dođe red na njega, a da korisnik koji radi za računalom to ne primijeti jer su promjene brze.

Mrežno povezivanje

Vrlo važna komponenta svih tipova računala je povezivanje s drugim uređajima. To povezivanje se odvija preko mrežnih protokola te sklopovlja i aplikacija koje ih koriste. Na taj način možemo komunicirati i dijeliti datoteke preko mreže. U toj komunikaciji mogu sudjelovati različiti uređajima kao što su na primjer prijenosnici, mobilni uređaji, pisači ili skeneri. Dva važna tipa mrežnog povezivanje je ono preko kabela (žičanih ili optičkih) i bežično povezivanje.

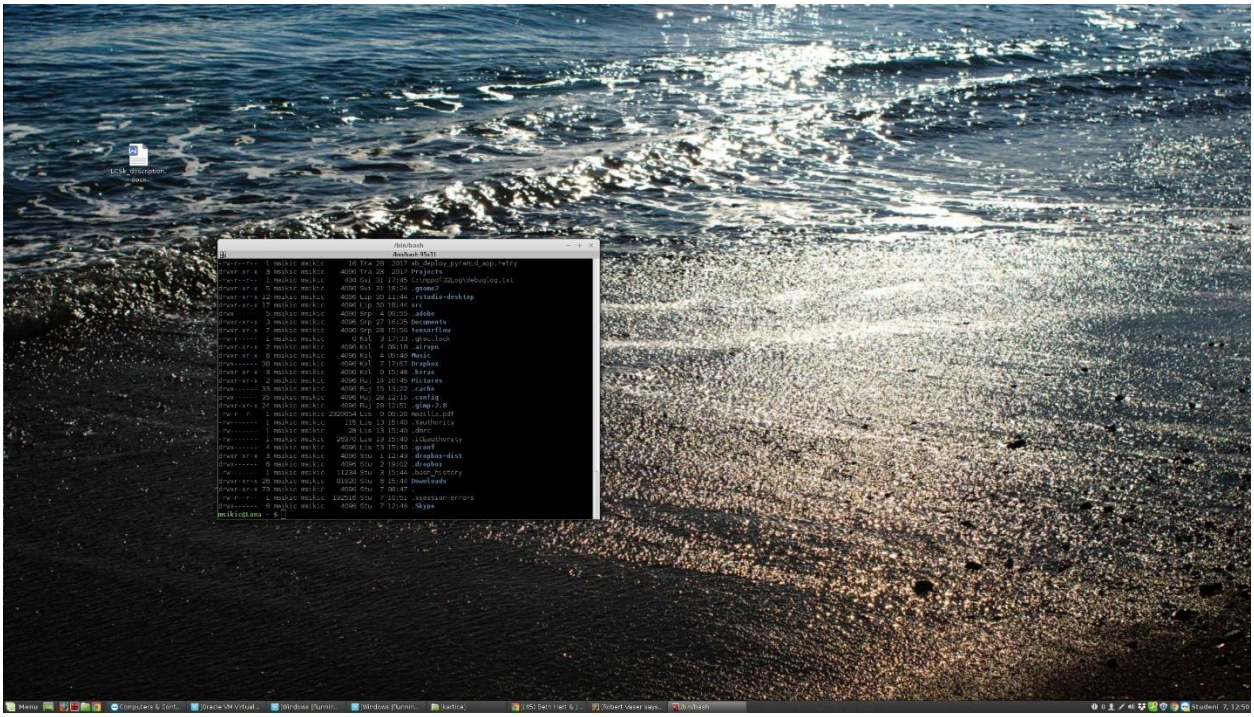
Sigurnost

Povezivanje uređaja u mreže povećava njihovu izloženost različitim sigurnosnim prijetnjama stoga operacijski sustav mora moći razlikovati pristup koji treba biti dozvoljen i procesiran od onih koji ne smiju biti procesirani. Najčešće se to obavlja na način da svi korisnici nemaju ista prava pri čemu su neki privilegirani. Stoga se kod korištenja računala razlikuju korisnici koji imaju administrativne ovlasti od „običnih“ korisnika. Na taj način se štite vitalni resursi

sustava. Najjednostavniji način autentifikacije korisnika je korištenje korisničkog imena i zaporke, no često se autentifikacija obavlja preko magnetskih ili čip kartica odnosno biometrijske provjere poput otiska prsta.

Korisničko sučelje

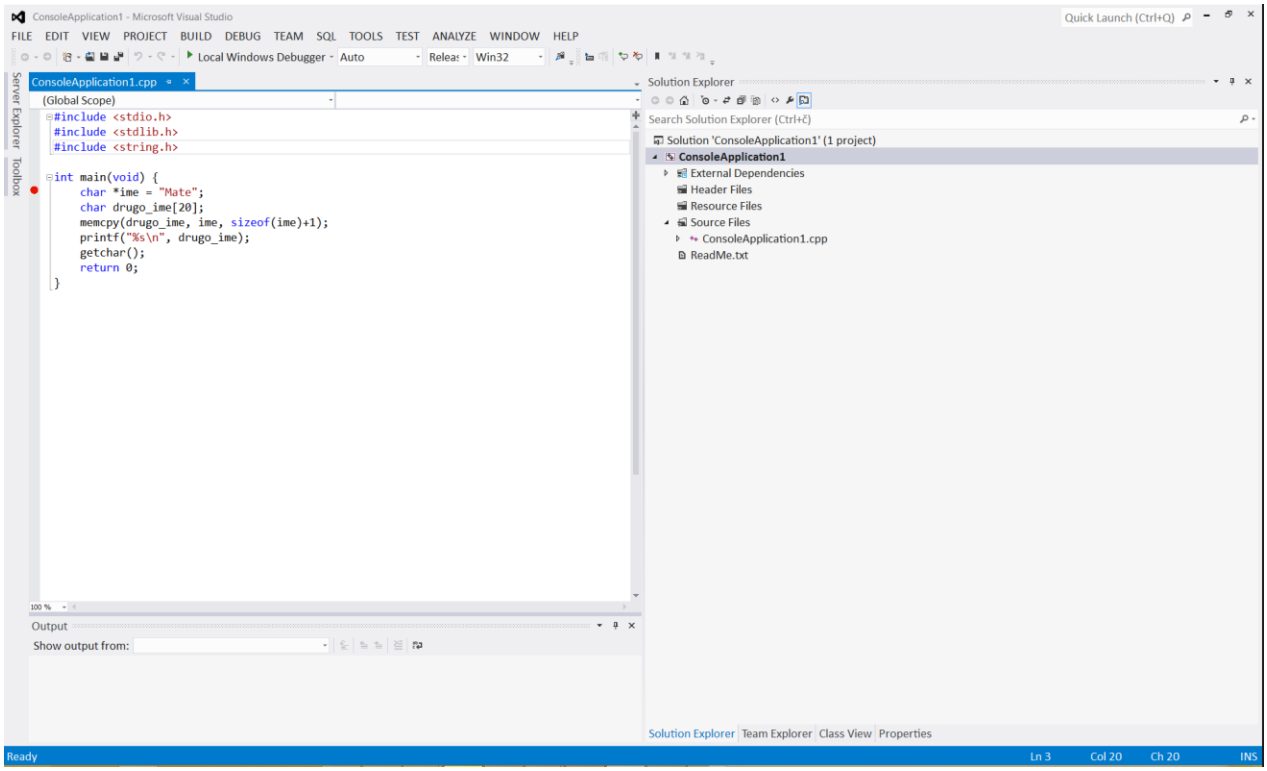
Operacijski sustav nam omogućava i sučelje za komunikaciju s računalom. Temeljna dva sučelja su grafičko sučelje i naredbena linija. U grafičkom sučelju (engl. graphical user interface - GUI) se za pokretanje i zatvaranje pojedenih programa i prebacivanje podataka koristimo većinom mišem ili zaslonom osjetljivim na dodir, dok kod naredbene linije sva komunikacija se odvija preko tipkovnice. Iako je komunikacija preko grafičkog sučelja jednostavnija, izdavanje naredbi preko naredbene linije omogućava veću fleksibilnost i preciznost te su nam dostupne operacije na nižoj razini. Danas se rijetko koristi isključivo naredbena linija osim za poslužitelje pa ako je potrebno njeno korištenje obično se podigne kao zasebni prozor unutar grafičkog sučelja.



Slika 2.2 Primjer Linux grafičkog sučelja unutar kojeg je otvorena naredbena linija kao zaseban prozor

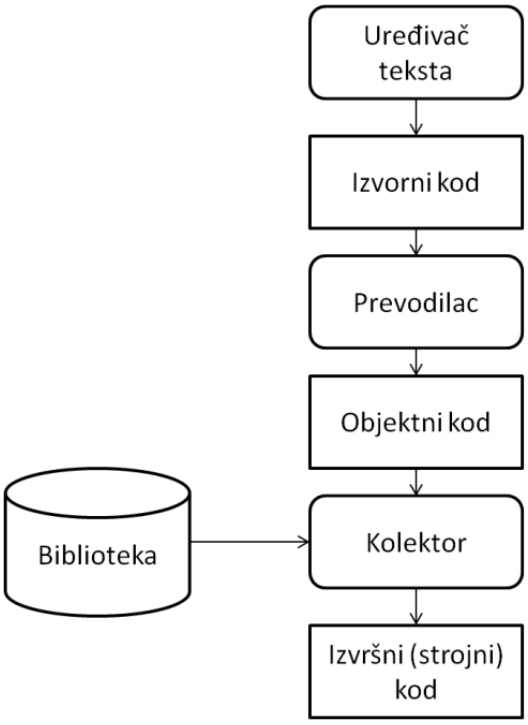
2.3 Pisanje i prevođenje softvera

Kada izrađujemo softver to možemo raditi u običnom uređivaču teksta ili to možemo raditi u specijaliziranim razvojnim okolinama koje su prilagođene pojedenim programskim jezicima i koje olakšavaju izradu koda i ispravljanje pogreški. Takve okoline imaju i sve alate potrebne za pretvorbu u izvršni kod. Primjer razvojnih okolina su Eclipse i Microsoft Visual Studio (Slika 2.3). Kao što smo ranije napomenuli danas većinom pišemo kod u višim programskim jezicima pri čemu je onda potrebno prije upotrebe kod prevesti u strojni jezik razumljiv računalu Slika 4 pokazuje osnovne faza prevođenja. Nakon što smo u nekom od uređivača teksta napisali kod taj kod se naziva izvornim kodom. U slijedećem koraku taj kod ide u prevođenje. Rezultat prevođenja alata kojeg nazivamo prevodilac ili kompajler je objektni kod. Objektni kod je često isti ili sličan strojnom kodu. Prilikom prevođenja prevodilac provjerava da li smo kod sintaksno dobro napisali, odnosno da li smo sve ključne riječi ispravno napisali i pratili sintaksna pravila za pojedini jezik. Na primjer u C jeziku će se kraj svake naredbe završavati s točka zarezom. Prevodilac će nam javiti da li je sve prošlo u redu i da li i gdje trebamo napraviti izmjene.



Slika 2.3 Primjer pisanja softvera u Windows Visual studio radnoj okolini

Slijedeći korak je povezivanje dobivenog objektnog koda s kodom biblioteka u kojima su opisane neke od često korištenih funkcija u danom programskom jeziku i koje ne moramo ponovo pisati svaki put ili povezivanje koda koji se nalazi u više datoteka. Taj korak obavlja alat koji se naziva kolektor (*engl.* linker). Kolektor će provjeriti da li je moguće cijeli kod dobro povezati te da li je dostupan kod za sve biblioteke koje smo naveli da ćemo koristiti. Vrlo često se dogodi da koristimo funkcije iz nekih biblioteka koje nemamo na računalu pa je onda potrebno ih pronaći i instalirati. Nakon uspješnog povezivanja imamo izvršni strojni kod koji se može pokretati na računalu. Vrlo često će taj kod raditi, ali ne i ispravno. Stoga je potrebno pripremiti odgovarajuće testne podatke za koje znamo kakav rezultat trebaju dati da možemo provjeriti ispravnost koda. Ukoliko dobivamo netočne rezultate u kodu imamo tzv. logičke pogreške i trebamo ih ispraviti. Cijeli proces pisanja koda, prevođenja, testiranja i ispravljanja greški jer iterativan. Proces ispravljanja pogreški u bilo kojoj fazi se često kolokvijalno naziva debugiranje (*engl.* debug).



Slika 2.4 Postupak izrade i prevođenja koda

2.4 Tipovi podataka

Računala spremaju podatke u obliku niza jedinica ili nula, no kada čitamo te podatke iz memorije možemo ih drugačije interpretirati, zavisno o njihovom tipu, npr. nekad ćemo ih interpretirati kao broj, a nekad kao slovo ili niz slova. Općenito postoji nekoliko osnovnih tipova podataka. To su:

- Cjelobrojni
- Tipovi u notaciji pomičnog zareza za prikaz decimalnih brojeva
- Logički
- Znakovni
- Alfanumerički nizovi

Za prikaz svakog od tipova podataka za svaki programski jezik zasebno definira se duljina u oktetima.

Cjelobrojni tip služi za prikaz pozitivnih i negativnih brojeva iz skupa cijelih brojeva. Važno je primijetiti da možemo prikazati samo određeni broj iz skupa cijelih brojeva, a ograničenje predstavlja duljina memorije koju smo rezervirali za pohranu. Kako za neke operacije ne zahtijevaju rad s velikim brojevima, dok nekima je ključan rad s njima postoji obično nekoliko tipova cjelobrojnih tipova podataka s više ili manje rezerviranih okteta. Negativne cijele brojeve pohranjujemo u aritmetici dvojnog komplementa kako bi olakšali računanje s njima. Kako je raspon cijelih brojeva ograničen moramo voditi računa da ne pokušamo pohraniti broj izvan raspona jer će se krivo prikazati.

Za prikaz realnih brojeva koristimo notaciju pomičnog zareza koja nam omogućava velik raspon brojeva koje spremamo, puno veći nego kod prirodnih brojeva no ukupno možemo pohranit u istoj količini memorije jednak broj cijelih i realnih brojeva. Kako je mali broj od beskonačnog skupa realnih brojeva koje možemo prikazati velik broj njih ćemo prikazati s greškom. Zavisno o količini dodijeljene memorije imat ćemo više ili manje precizan prikaz realnih brojeva. Slično kao i kod cjelobrojnih tipova i ovdje imamo nekoliko tipova podataka veće ili manje preciznosti zavisno o potrebi. Pogotovo pri preciznim računanjima treba voditi da li nam je preciznost koju nam garantira odabrani tip podataka dovoljna.

Logički tip podataka je zadužen za spremanje logičkih istina i neistina. Zavisno o programskom jeziku može se realizirati kao zaseban tip ili se za njegov prikaz koristi cjelobrojni tip (npr. u programskom jeziku C).

Osim brojeva i logičkog tipa podataka važno nam je mogućnost pohranjivanja tekstova odnosno pojedinih znakova. Za to se u jezicima definira posebni znakovni tip podataka. Nizovi znakova mogu biti pohranjeni kao niz uzastopnih memorijskih lokacija na kojima su pojedini znakovi ili se definira poseban tip. Osnovni kod prikaz slova engleske abecede, znamenaka i raznih kontrolnih znakova kao što su razmaci se naziva ASCII (*engl.* American Standard Code for Information Interchange). ASCII kodom je definirano 256 znakova (jedan oktet). Kako mnoge abecede, poput i naše, koriste znakove koji nisu dio engleske abecede u slučaju pokušaja prikaza takvih znakova dobit ćemo čudne znakove na zaslonu ili pisaču. Zbog toga razloga se izbjegava pisanje dijakritičkih znakova u imenima datoteka. Uočivši probleme sa ASCII znakova kasnije su razvijeni novi kodovi poput UTF-8 ili UTF-16 koji omogućavaju pisanje slova iz velikog broja različitih abeceda.

3 Osnove programiranja kroz programski jezik C

Kao programski jezici na ovom predmetu odabrani su C i C++. C++ je napravljen na temelju C-a pri čemu su dodana mogućnost korištenja drugačijih paradigmi (npr. objektno programiranje). U ovom prvom dijelu fokusirat ćemo se isključivo na C jezik. C jezik je važan jezik kojim je pisan velik broj programa niske razine. Iako su danas neki drugi jezici popularniji od njega, on omogućava dobro razumijevanje što se događa na nižim razinama, vrlo je brz i ima relativno jednostavnu sintaksu. U jezicima C i C++ je napisana većina sistemskog softvera.

3.1 Jednostavan C program

Kôd 3.1 prikazuje jednostavan C program koji ispisuje na ekran rečenicu „Moj prvi C program!!“. Ovaj kod je prikladan za objašnjene sintakse C jezika.

```
#include <stdio.h>
int main()
{
	// funkcija printf() prikazuje niz znakova unutar navodnika
	printf("Moj prvi C program!!");
	getchar();
	return 0;
}
```

Kôd 3.1 Jednostavan primjer C kôda

Svaki C program obavezno mora imati `main` funkciju (označeno plavo). Svaka funkcija u C-u mora imati naznačen tip podataka. U ovom slučaju je to integer (oznaka `int`) – cjelobrojni tip podataka. Unutar okruglih zagrada upisujemo argumente koje predajemo funkciji. Kako u ovom slučaju ne predajemo argumente unutar zagrada nemamo ništa navedeno. Posebno za `main` funkciju možemo naglasiti da ne predajemo argumenta sa ključnom riječi `void` koji smo mogli umetnuti unutar zagrada i koja označava da nema

argumenata. Početak i kraj tijela funkcije, odnosno ono što funkcija izvršava u C-u se označava s vitičastim zagradama (označeno plavo). Crveno označeni dio, koji se obavezno nalazi izvan tijela funkcija, nam označava koju biblioteku želimo uključiti. Za uključivanje koristimo naredbu `#include`. S obzirom da u programu koristimo već gotovu funkciju `printf`, kojom ispisujemo tekst unutar navodnika na ekran, trebamo naglasiti da ćemo koristiti standardnu biblioteku za ulaz i izlaz `stdio.h` u kojoj se ta funkcija nalazi. Nakon funkcije `main` se nalazi funkcija `getchar`. Ona je isto dio `stdio.h` biblioteke i funkcija zaustavlja izvršavanje programa dok god se ne unese neki znak. Ta nam funkcija nije trebala u ovom programu, ali kada bi ga pokrenuli poruka bi prekratko bila na zaslonu tako da ju vjerojatno ne bi stigli pročitati. Stoga nama funkcija `getchar` daje mogućnost da pročitamo i unošenjem bilo kojeg znaka program će završiti. Funkcija `main` završava povratkom, odnosno naredbom `return`. Kod funkcije `main` uvijek vraćamo 0. Zeleno je u kodu označen komentar koji se označava sa dva kose crte na početku retka. Komentar se ne prevodi u strojni jezik već služi programeru za pojašnjenje koda. Nakon nekoliko mjeseci ili kada netko drugi preuzme kod teško je protumačiti što točno određeni dio koda radi stoga je važno pisati komentare. Ako želimo komentar koji se proteže kroz nekoliko redaka onda ga započinjemo sa `/*`, a završavamo na kraju teksta sa `*/`.

Ovaj program smo mogli napisati i bilo kojem tekstualnom editoru (npr. Notepad ili Notepad++ kod MS Windowsa ili nano, vi ili neki tekstualni editor s grafičkim sučeljem kod Linux operacijskog sustava) i spremiti kao `ime_datoteke.c` ili u nekom od razvojnog okruženja. U slučaju C++ programa datoteku spremamo kao `ime_datoteke.cpp`. Nakon toga potrebno je pokrenuti prevoditelj koji je sastavni dio razvojnog okružja ili ako smo pisali u tekstualnom editoru vanjski prevoditelj poput gcc-a.

3.2 Tipovi podataka u programskom jeziku C

Podatke koje spremamo možemo podijeli u one koji su promjenjivi i one koji su nepromjenjivi i koje nazivamo konstantama. Svi podaci se spremaju u memoriju. Da bi smo lakše baratali podacima dajemo im oznake. Te oznake se nazivaju varijablama i predstavljaju naše ime za lokaciju u memoriju. Kada prvi put uvodimo varijablu obavezno moramo navesti kojeg je tipa. Na taj način će računalo protumačiti niz jedinica i nula spremljenih na određenoj memorijskoj lokaciji. Npr. naredba

```
int a;
```

će stvoriti novu varijablu tipa integer koja pokazuje na određenu lokaciju u memoriji. Važno je napomenuti da nakon gornje naredbe varijabla `a` ima neku vrijednost, no ta nam vrijednost nije poznata u trenutku izvođenja programa i svaki put kada pokrenemo računalo ta vrijednost će se promijeniti.

U programsku jeziku C koristimo ove osnovne tipove podatka:

- Cjelobrojni (`short`, `int` i `long`).
- Realni (`float`, `double`, `long double`).
- Znakovni (`char`).

Za svaki od ovih tipova podataka definirana je veličina memorije koju zanima. Dok je su `float` i `double` precizno definirani s 32 odnosno 64 bita za `int` (kratica za integer) to ne znamo dok ne provjerimo da li se na našem računalu ponaša kao 32 bitni ili 64 bitni. Za potrebe objašnjenja koristit ćemo pretpostavku da se radi o 32 bitnom integeru. Za short se obično uzima da je 16 bita dok za `long` je definirano da je veličine `int` ili veći. Za `long double` analogno definiramo da je duljine kao `double` ili dulji.

Znakovni niz `char` je duljine 8 bita i znakovi se spremaju prema ASCII kodu (http://www.asciitable.com/). Kada želimo spremiti u varijablu tipa char neku vrijednost to možemo na dva načina. Jedan način je da znamo o kojem ASCII znaku je riječ, a drugi je da unesemo sam zna. To možemo napraviti na slijedeći način:

```
char prvi_znak = 65;
char drugi_znak = 'A';
```

U oba slučaja smo u varijable unijeli znak A, ali u prvom slučaju smo ga unijeli preko njegove ASCII vrijednosti, a u drugom smo ga eksplicitno unijeli. Važno je da kada unosimo sam znak da moramo staviti u jednostruke navodnike.

Važno je poznavati duljinu zapisa za pojedini tip podataka jer onda znamo koji raspon brojeva možemo prikazati. Za npr. 32-bitni `int` to je $-2\ 147\ 483\ 648$ (-2^{31}) do 2 147 483 647 ($2^{31}-1$). Svi tipovi podataka podrazumijevaju i pozitivne i negativne brojeve. Ukoliko želimo povećati raspon samo pozitivnih brojeva jer znamo da nećemo imati negativne brojeve onda stavimo ispred tipa podataka modifikator `unsigned` što označava da se radi samo o pozitivnim brojevima. Npr. `unsigned int` je tip podataka koji ima raspon 0 - 4294967296 (2^{32}). Kada napišemo npr. samo `int` ili `float` podrazumijeva se da se radi o `signed int` i `signed float`, ali `signed` ne treba eksplicitno napisati.

Ako nismo sigurni kolika je veličina npr. int na našem računalu, to možemo provjeriti sa `sizeof` naredbom

```
sizeof(int); // veličina u bajtovima
```

Sa `sizeof` naredbom možemo provjeriti veličinu memorijske lokacije na kojoj je neka varijabla zapisana.

U programskom jeziku C nije eksplicitno definiran logički tip podataka, već kada radimo logičke operacije onda koristimo sve ostale tipove podataka i to na način da je logička 0 samo 0 u bilo kojem tipu, a sve ostalo je logička jedinica neovisno da li je broj pozitivan ili negativan. Npr. ako imamo broj 0.0000 `float` tipa onda je to logička 0, no već 0.000001 je logička jedinica. Naknadno je u C99 standard dodan tip `_Bool` kao logički tip podataka, ali mi ga nećemo u ovim materijalima koristiti. U C++ jeziku za postoji `bool` tip podataka za logičke operacije.

Osim varijabli imamo i nepromjenjive podatke koje nazivamo konstantama konstante, vrijednosti koje se ne mijenjaju tijekom izvođenje programa. One su spremljene i u specijalni dio memorije. Ako imamo ovaj niz od dvije naredbe

```
int a;
a = 3;
```

prvo u memoriji stvorimo varijablu `a` tipa integer koja pokazuje na neku memorijsku lokaciju i čiju vrijednost ne znamo. Nakon toga na tu memorijsku lokaciju upišemo vrijednost 5 to je ostvareno operatorom pridruživanje =. Važno je naglasiti da ovdje 5 je konstanta i da se ta vrijednost ne može promijeniti. Kao i kod varijabli i konstante mogu imati različite tipove podataka. Tako npr. kada napišemo 5 prevoditelj zna da je tip ove konstante `int`. Da smo napisali 5. onda bi tip varijable bio `double`. Ukoliko bismo eksplicitno željeli da nam je tip podatka konstante `float` ili `long` to bismo trebali posebno naglasiti pa bismo napisali `5.f` odnosno `5l`.

Ispis podataka

U prvom programu smo koristili funkciju `printf` iz standardne `stdio.h` biblioteke. U tom programu ispisali smo unaprijed definirani tekst. Što ako trebamo ispisati sadržaj pojedine varijable. Ako imamo varijablu i u kojoj je pohranjen cijeli broj tu varijablu ćemo ispisati naredbom

```
int i = 5;
printf("Vrijednost varijable i je: %d.\n", i);
```

Ovaj isječak koda će ispisati tekst: `Vrijednost varijable i je: 5.` U gornjem primjeru funkcija ima dva argumenta. Prvi argument je tekst unutar dvostrukih navodnika u kojem zadajemo format ispisa.

Drugi je sama varijabla koju ćemo koristiti. Ako ima više varijabli, bit će odvojene zarezima. Prvi argument isto tako možemo podijeliti ugrubo u dva dijela. Crveno je označen tekst koji će pisati i u ispisu, a plavo su označeni specifični specijalni znakovi. Specijalni znak `%d` je modifikator da u tome dijelu treba doći cjelobrojna vrijednost. Ovaj modifikator se sastoji od dva dijela, znaka `%` što najavljuje da će biti definiran tip podataka koji će se umetnuti i znaka d koji najavljuje da će taj tip biti cijeli broj. Specijalni znak `\n` je oznaka da će nakon toga se ispisati prelazak u novi red. Ovaj specijalni znak može biti bilo gdje u tekstu i upravo će na tom mjestu biti prelazak u novi red. Problem nastaje kada želimo da imamo u tekstu specijalne znakove poput \ ili `%`. Tada ispred njih stavimo znak \ kojim dajemo naredbu prevoditelju da slijedeći znak ne smatra specijalnim znakom.

Tablica 3.1 prikazuje osnovne modifikatore za printf i scanf naredbe.

Unos podataka

Slično naredbi `printf` za unos podataka koristimo naredbu `scanf`. Ako želimo unijeti jedan cijeli broj možemo napisati ovaj niz naredbi `printf` i `scanf`

```
int x;
printf("Unesi cijeli broj: ");
scanf("%d", &x);
```

Slično kao i kod `printf` naredbe kod `scanf` naredbe imamo na početku izraz pod navodnicima u kojem unosimo modifikatore odnosno kažemo računalu koji tip podataka treba očekivati. Nakon toga slijedi niz pozicija na koje unosimo vrijednosti. Ovdje je važno primijetiti da za razliku od `printf` naredbe ovdje imamo operator `&` ispred varijable `x`. Taj operator nazivamo operatorom adrese i kaže nam na koju memorijsku lokaciju unosimo vrijednost. Zbog toga je potrebno unaprijed imati deklariranu varijablu kojoj je dodijeljena memorijska lokacija.

Tablica 3.1 Modifikatori za printf i scanf naredbe

Kod	Tip podataka	Format zapisa
d	int	Dekadski cijeli broj
c	char	Jedan znak
s	string	Niz znakova
f	float	Realni broj
lf	double	Realni broj

3.3 Operacije nad podacima

U prethodnom poglavlju spomenuli smo osnovnu operaciju, pridruživanje. Gornja dva reda smo mogli spojiti u jedan i napisati

```
int a = 3;
```

To znači da smo stvorili novu varijablu koju zovemo `a`, koja je tipa `int` i memorijska lokacija koja joj je pridružena sada sadrži broj 3. Kada na ovaj način prvi put pridružimo varijabli vrijednost to nazivamo inicijalizacija.

3.3.1 Aritmetički operatori

Aritmetički operatori su važna skupina operatora. Tablica 3.2 prikazuje osnovne aritmetičke operatore.

Naredbu

```
a = a + 3
```

možemo gledati kroz dva koraka. U prvom koraku se vrijednost koja je spremljena u `a` uvećava za 3, a u slijedećem koraku se ta nova vrijednost sprema ponovo u `a`. Ako je u prethodnom koraku u a bila vrijednost tri sada je nova vrijednost 6.

Tablica 3.2 Prikaz osnovnih matematičkih operatora

Aritmetički operator	Značenje operatora
+	Zbrajanje
-	Oduzimanje
*	Množenje
/	Dijeljenje
%	Ostatak cjelobrojnog dijeljenja (još se naziva modulo ili modulus)

Pri korištenju aritmetički operatora treba voditi računa o tipovima varijabli i konstanti koje koristimo. Kada koristimo aritmetičke izraze i tip `char` se ponaša kao cjelobrojni tip. Pravilo je kada imamo dva operanda (neovisno da li su varijable ili konstante) istog tipa rezultat je istog tipa. Kada su vrijednosti različitog tipa onda koristimo slijedeća pravila:

1. Ako je jedan od operanada tipa `long double`, preostali operand se pretvara u tip `long double`
2. Ako je jedan od operanada tipa `double`, preostali operand se pretvara u tip `double`
3. Ako je jedan od operanada tipa `float`, preostali operand se pretvara u tip `float`
4. Ako je jedan od operanada tipa `long`, preostali operand se pretvara u tip `long`
5. Operande tipa `short` i `char` pretvoriti u tip `int`

U slučaju kada se u izrazima pojavljuju `unsigned` tipovi podatka, pravila su složenija i ovise o implementaciji.

Kao primjer za važnost vođenja računa o tipovima podataka pretpostavimo da imamo ovaj slijed naredbi

```
int a = 3;
float b;
b = a / 2;
```

Prvo stvorimo (deklariramo) varijablu `int` varijablu `a` i inicijaliziramo ju vrijednost na 3, nakon toga stvorimo varijablu `b` tipa `float`. Na kraju podijelimo vrijednost `a` s dva i pridružimo `b`. Važno je napomenuti da se prvo odradi do kraja operacija `a / 2` i tek nakon toga se odradi operacija pridruživanja. Primijetimo da je 2 konstanta koja je tipa `int`. Kako je i a int, imat ćemo cjelobrojno dijeljenje. Cjelobrojno dijeljenje funkcionira na način da se nakon dijeljenja kao rezultat dobije cijeli broj pri čemu je to najveći cijeli broj od onoga što bi dobili od rezultata dobivenog dijeljenjem realnih brojeva. U ovom slučaju 3 / 2 je 1.5 i rezultat je 1! Kada smo dobili taj rezultat njega produžujemo u varijablu `b` tipa `float`. Vrijednost u b je sada `1.0f` (f je oznaka za `float`).

Cjelobrojno dijeljenje u slučaju kada smo željeli napraviti dijeljenjem realnim brojevima je jedna od najčešćih pogrešaka i treba jako voditi računa o tome. Postoji dva načina na koja

smo to mogli izbjeći. Prvi je da nam konstanta nije bila tipa `int` nego `float` ili `double`. Dakle, mogli smo napisati

```
b = a / 2.
```

pri čemu točka iza 2 označava da se radi o tipu `double`. Drugi način je da radimo eksplicitnu promjenu tipa podataka. Ako smo npr. željeli samo u ovoj operaciji vrijednost koja je pohranjena u `a` promijeniti u `float` tip napisali bi

```
b = (float) a / 2;
```

Kao u matematici i kod aritmetičkih operatora trebamo voditi računa o prioritetu pojedinih operatora. I ovdje množenje i dijeljenje te ostatak cjelobrojnog dijeljenja imaju veću prioritet od zbrajanja i oduzimanja. Ako želimo promijeniti poredak operacija moramo primijeniti okrugle zagrade.

Kada zapisujemo matematičke operacije možemo koristiti kraće izraze. Tako je npr. izraz

```
a = a / 5;
```

možemo napisati

```
a /= 5;
```

Iz ovoga izraza se vidi da prvo se koristi operator dijeljenja pa nakon njega operator pridruživanja. Isti način se može primijeniti za sve ostale aritmetičke operatore.

Dva posebna operatora su takozvani inkrement (++) i dekrement (--) operatori koji mogu uvećati ili umanjiti vrijednost varijable za 1. Za svaki od njih imamo dvije verzije pre i post. Tako na primjer izrazi

```
i++; // pre-inkrement i = i +1
++i; // post-inkrement i = i +1
```

će uvećati vrijednost varijable i za 1. Međutim, kada imamo druge izraze u jednadžbi moramo voditi računa o da li se radi o pre ili post. Za primjer pretpostavimo da imamo ovaj niz naredbi

```
i = 1;
j = ++i;
```

U ovom slučaju imamo pre-inkrement operator koji će odmah uvećati iznos varijable `i`, a tek nakon toga izvršiti pridruživanje. Zbog toga nakon završetka ovih operacija u obje varijable bit će spremljena vrijednost 2. Ako samo pre-inkrement s post-inkrement operatorom

```
i = 1;
j = i++;
```

dobit ćemo drugačiji rezultat. U ovom slučaju će se napraviti sve ostale operacije, a tek na kraju će se uvećati vrijednost varijable i tako da će vrijednosti varijabli `i` i `j` ovaj puta biti 2 i 1.

Slično imao i operatore - - za umanjivanje vrijednosti za 1. Treba biti jako oprezan pri korištenju ovih operatora i izbjegavati presložene izraze.

3.3.2 Relacijski operatori

Kada želimo utvrditi odnos među operandima koristimo relacijske operatore. Nakon uvrštavanja vrijednosti operanada u izraz dobivamo logičku istinu ili laž. Tablica 3.3 prikazuje relacijske operatore.

Kada želimo usporediti da li su dva operanda jednako koristimo == za razliku od operacije pridruživanja kada koristimo operator =. Npr. ako provjeravamo da li varijable a = 1 i b = 2 sadrže iste brojeve, rezultat operacije a == b bit će logička laž (0), no kada bi napisali a = b zapravo bi pridružili iznos iz b u a i rezultat operacije bi bio broj koji pridružujemo (2) što je logička istina. Dodatno u a bi sada bio broj 2 umjesto 1. Ova pogreška spada u jednu od najčešćih pogrešaka pri programiranju u jeziku C.

Tablica 3.3 Relacijski operatori

Relacijski operator	Značenje
<	manje
>	veće
<=	manje ili jednako
>=	veće ili jednako
==	jednako
!=	različito

3.3.3 Logički operatori

Često nam nije dovoljno da usporedimo dva operanda nego bi htjeli imati složene relacijske izraze za koje koristimo logičke operator `I`, `ILI` i `NE`. Tablica 3.4 prikazuje osnovne logičke operatore.

Tablica 3.4 Logički operatori

Relacijski operator	Značenje
&&	Logičko I
‖	Logičko ILI
!	Logičko NE

Dok su operatori logičko I i logičko ILI binarni operatori, odnosno djeluju na dva operanda, operator logičko NE je unarni operator koji djeluje na samo jednog operanda i to onoga njemu s desne strane (nalazi se ispred operanda).

Primjer složenog relacijskog izraza je `x > 20 && x < 50` kojim želimo utvrditi da li se `x` nalazi unutar intervala (20, 50). Česta pogreška u ovakvim slučajevima je da zaboravimo u drugom izraz `x` pa napišemo `x > 20 && < 50` što će dovesti do pogreške u prevođenju.

3.3.4 Operatori nad bitovima

Operatori nad bitovima (Tablica 3.5) su poseban tip operatora definiran nad bitovima pomoću kojih je zapisana vrijednost varijable. Kao i kod logičkih operatora mogu biti binarni ili unarni.

Uzmimo dva cjelobrojna zapisa 00001000 (dekadski 8) i 00001111 (dekadski 15) i napravimo s njima operacije s operatorima nad bitovima. Tablica 3.6 prikazuje primjer operacija s ova dva cijela broja. Sve operacije se odnosno na svaki bit zasebno.

Operacija isključivo ILI se razlikuje od ILI po tome da bi oba trebaju biti različita da bi rezultat bio logička istina (1). Iako smo cjelobrojne brojeve napisali u binarnom zapisu, isto vrijedi kada u programu koristimo dekadski zapis.

Tablica 3.5 Operatori nad bitovima

Operator	Značenje	Tip operanada
&	I	Binarni
|	ILI	Binarni
^	Isključivo ILI	Binarni
~	NE	Unarni
<<	Posmak u lijevo	Binarni
>>	Posmak u desno	Binarni

Tablica 3.6 Primjer operacija s binarnim operatorima

Izraz	Operator	Rezultat binarno	Rezultat dekadski
00001000 & 00001111	Binarno I	00001000	8
00001000 | 00001111	Binarno ILI	00001111	15
00001000 ^ 00001111	Birano isključivo ILI	00000111	7
~ 00001111	Binarno NE	11110000	-16 (aritmetika dvojnog komplementa)
00001111 << 2	Posmak u lijevo 2 mjesta	00111100	60 (kao da smo množili s 4)
00001000 >> 2	Posmak u desno 2 mjesta	00000010	2 (kao da smo podijelili s 4)

Kada radimo posmak na desno ili lijevo s unsigned tipovima podataka ako neki brojevi odu lijevo ili desno izvan zapisa oni se više ne računaju, a na suprotnoj strani dodajemo nule. Npr. ako broj 8 zapisan kao gore pomaknemo za 5 mjesta u lijevo ili desno dobit ćemo kao rezultat 0. Standard C jezika ne objašnjava detaljno što se događa s negativnim brojevima kada imamo signed tipove podataka i rezultat izvršavanja na različitim računalima može se razlikovati.

3.3.5 Kontrola toka podataka

U programima često imamo potrebu raditi različita grananja ovisno o logičkom izrazu ili imamo potrebnu ponavljati niz naredbi određeni broj puta (petlja).

Naredbe grananja

U jeziku C osnovno grananje se radi naredbom `if`. Zamislimo da za svakog studenta znamo koliko je bodova od 0 do 100 dobio na ispitu i na osnovu toga želimo mu upisati ocjenu. Za broj bodova koristimo varijablu `bodovi`, a za ocjenu varijablu `ocjena`.

Kod za to bi mogli napisati ovako:

```
if (bodovi >= 90) {
      ocjena = 5;
}
else if (bodovi >= 80) {
      ocjena = 4;
}
else if (bodovi >= 65) {
      ocjena = 3;
}
else if (bodovi > = 50 {
      ocjena = 2;
}
else {
```

```
    ocjena = 1;
}
```

Ako pogledamo kod možemo vidjeti nekoliko različitih naredbi (`if`, `else if` i `else`). Primijetite da svaka od naredbi ima isti oblik

```
Naredba (uvjet) {
    // Blok naredbi
}
```

Izuzetak je naredba `else` koja nema poseban uvjet već pokriva sve preostale slučaje. Naredbe se izvršavaju jedna po jedna od gore prema dole. Ako je neki uvjet logički istinit izvršit će se blok naredbi unutar vitičastih zagrada i nakon toga se preskaču preostale naredbe u grananju i izvršava se naredba koja dođe iza kraja grananja. Kod grananja, blok naredbi se uvlači da bi se dobila preglednost i lakše uočilo koji blok pripada kojem grananju. U grananju jedina obavezna naredba je `if`. Ukoliko nema potrebe za time ostale naredbe `else if` i `else` se ne moraju pisati. `else if` koristimo kada želimo specificirati još jedan ili više posebnih slučajeva.

Ukoliko se blok naredbi unutar vitičastih zagrada sastoji od samo jedne naredbe onda možemo te vitičaste zagrade izostaviti za taj specifičan blok.

U gornjem kodu smo mogli preciznije specificirati raspon za svaku ocjenu, no kako se naredbe izvršavaju po redu nema potrebe za time. Npr. ako prvo provjerimo da li je student dobio više od 90 bodova za ocjenu 5, ne treba za ocjenu provjeravati da li je u između 80 i 90 bodova već je dovoljno da je 80 i više.

Drugi način na koji možemo ostvariti grananje je pomoću `switch` i `case` naredbi. Ovo grananje je pogodno kada želimo ostvariti grananje na nekom cjelobrojnom izrazu. S obzirom da i znakovi spadaju u cjelobrojni tip podataka može se koristi i za njih. Kao primjer uzmimo da provjeravamo koji aritmetički operator se nalazi u varijabli operator. To možemo napraviti kao u Kôd 3.2. Kod ovoga grananja isprobavamo različite slučajeve (*engl.* case) sve dok ne dođemo do ispravne. Vrlo važnu ulogu imaju naredbe `break` na kraju svakog `case`. Da ih nema kod bi se nastavio na slijedeći `case`. U slučaju naredbe `break` idemo na kraj

cijeloga `switch case` bloka naredbi i nastavljamo s prvom naredbom nakon njega. To nam nekada odgovara, ali vrlo često ne. Ako niti jedna od slučajan nije prisutan, opcionalno na kraj možemo staviti slučaj `default` koji poput `else` naredbe pokriva sve ostale slučaje.

```
switch (operator) {
case  '+':
      printf(„Zbrajanje“);
      break;
case  '-':
      printf(„Oduzimanje“);
      break;
case '*':
      printf(„Množenje“);
      break;
case '/':
      printf(„Dijeljenje“);
      break;
case '%':
      printf(„Ostatak cjelobrojnog dijeljenja“);
      break;
default:
      printf(„Pogrešan operator!“);
      break;
}
```

Kôd 3.2 Primjer sa switch case grananjem

Programske petlje

Kada želimo određeni blok naredbi ponoviti više puta koristimo programske petlje. Razlikujemo tri vrste petlji.

Prva vrsta petlji su `while` petlje kod kojih unaprijed ne znamo koliko puta će blok naredbi unutar petlje biti izveden, no može se dogoditi da neće biti izveden niti jedanput. Kod `while` petlji ispitivanje uvjeta je na početku petlje.

Druga vrsta petlji su `do while` petlje kod koji znamo da će blok naredbi biti izveden barem jedanput. Ispitivanje uvjeta je na kraju petlje.

Treća vrsta su `for` petlje koje se koriste u slučajevima kada točno znamo koliko puta ćemo neki blok naredbi izvršiti.

While petlja

Jednostavan primjer za `while` petlju je slijedeći

```
int i = 1; // inicijalizacija
while (i < 5) { // provjera uvjeta
      printf(„Broj je manji od 5!“);
      i = i +1; // uvecanje
}
```

U ovom slučaju mi smo unaprijed inicijalizirali vrijednost varijable `i`, ali to se moglo promijeniti tijekom programa. Kao što možemo vidjeti ova petlja će se izvršavati sve dok je uvjet ispunjen, odnosno kada je i manji od 5. U trenutku kada postane 5 uvjet neće biti ispunjen i petlja će se prekinuti. U `while` petlji je nužno imati naredbu u kojoj mijenjamo vrijednost varijable koju provjeravamo na početku. Da toga nema petlja bi se mogla beskonačno izvršavati. Slično kao i kod grananja, blok naredbi koje se izvršavaju pišemo uvučeno radi preglednosti. Ukoliko imamo samo jednu naredbu u unutar bloka, nije potrebno koristiti vitičaste zagrade. Npr. ako smo samo uvećavali vrijednost, bez ispisa, mogli smo napisati

```
while (i < 5)
      i = i +1;
```

Do while petlja

Ako želimo da se uvjet izvrši barem jedanput koristimo `do while` petlju. Gornji primjer koristeći do while petlju bi mogli napisati kao

```
int i = 1; // inicijalizacija
do {
      if (i < 5)
            printf(„Broj je manji od 5!“) ;
      i = i + 1; // uvećanje
while (i < 5); // provjera uvjeta
```

Za razliku od prošle petlje ovdje smo zbog toga što se petlja mora izvršiti barem jedanput trebali dodatno provjeriti u bloku naredbi da li je broj već pri ulasku u petlju manji od 5. U suprotnom ga ne bismo ispitivali. Kod ove petlje je ispitivanje uvjeta na kraju. Slično kao i kod `while` petlje, ukoliko je samo jedna naredba u bloku vitičaste zagrade nisu potrebne.

For petlja

`for` petlje se koriste kada imamo točno određeni broj naredbi za izvršiti. Isti primjer bismo koristeći `for` petlju napisali kao:

```
int i;
for (i = 1; i < 5; i++) {
      Printf(„Broj je manji od 5!“);
}
```

Ako pogledamo izraz u zagradi, on se sastoji od tri dijela: (i) inicijalizacije, (ii) provjere uvjeta i (iii) uvećanja varijable koju koristimo za brojanje. Petlja funkcionira na način da prvi put kada uđemo u nju se napravi, inicijalizacija, nakon toga se provjerava uvjet. Ako je uvjet zadovoljen nastavlja se s blokom naredbi. Nakon završetka u ovom slučaju ćemo uvećati varijablu `i` za jedan te ponovo provjeravamo uvjet. Ovo se ponavlja sve dok uvjet nije

ispunjen. Primijetite da se `for` petlja može lako napisati kao `while` petlja kada ta dijela iz zagrade rasporedimo. U gornjem primjeru imamo samo jednu liniju koda unutar petlje pa smo mogli pisati bez vitičastih zagrada.

U ovom primjeru smo uvećali vrijednost varijable i za jedan, no vrlo često u petljama vrijednosti umanjujemo, množimo, dijelimo vrijednosti varijable koju ispitujemo. Postoje još brojni drugi načini ovisno o tipu varijable koju koristimo.

Break i continue naredbe

Naredbu `break` smo upoznali kod `switch case` grananja. Kada ju koristimo unutar petlje ona prekida izvršavanje petlje i nastavlja s linijom koda nakon petlje.

Vrlo često kada koristimo `while` petlju u situaciji kao što je ova

```
int a;
scanf(„unesi broj: %d", &a);
while (a != b) {
      // blok naredbi
      scanf(„unesi broj: %d", &a);
}
```

Problem s ovim kodom je što se isti `scanf` naredba ponavlja dva puta. Ovakav kod se naziva petlja i pol i izbjegava se. Pošto se `scanf` izvodi barem jedan put, trebalo je primijeniti `do while` petlju umjesto `while` petlje. U mnogim jezicima ne postoji do `while` petlja nego se u ovakvim situacijama koristi kombinacija beskonačne `while` petlje i `break` naredbe. Beskonačna `while` petlja je petlja koja bi da nema `break` naredbe unutar nikada se ne bi prekinula. To postižemo uvjetom koji će uvijek biti logička istina. Primjer takve petlje je ova

```
while (1) {
      \\ blok naredbi
}
```

Gornji primjer s beskonačnom petljom izbjegavajući petlju i pol možemo napisati kao

```
int a;
while (1) {
      scanf(„unesi broj: %d", &a);
      // blok naredbi
      if (a == b)
            break;
}
```

Naredba slična naredbi `break` je naredba `continue`. Za razliku od naredbe break koja znači automatski izlazak iz petlje, nakon naredbe `continue` izvršava se ponovo uvjet kod `do while` i `while` petlje, odnosno dio za uvećavanje i uvjet kod `for` petlje.

3.4 Polja, pokazivači i dinamička alokacija memorije

3.4.1 Polja

Zamislimo da želimo napraviti program u kojem ćemo prebrojati koliko puta nam se ponovio pojedini broj u rasponu od 0 do 99. Prva ideja je da napravimo 100 varijabli u kojima ćemo pratiti za svaki broj koliko se puta ponovio. Problem je raditi sa tako velikim brojem varijabli i kod bi bio jako dugačak jer bi trebao imati veliko grananje u kojem bi morali imati opciju za svaki broj.

U ovakvim slučajevima koristimo složenije strukture koje nazivamo polje. Polje je zamjena za više varijabli istog tipa koje zamislimo da su poredane jedna iza druge. Pristupamo im preko njihovog indeksa. U jeziku C indeksiranje članova počinje s 0 umjesto 1.

Primjer jednostavnog koda u kojem ćemo stvoriti polje od 10 cjelobrojnih brojeva i kojem ćemo u svaki element polja upisati indeks preko kojeg mu se pristupa.

```
int polje[10];
int i;
for (i =0, i < 10; i++)
     polje[i] = i;
```

U prvoj liniji koda smo stvorili polje od 10 varijabli tipa `int` i nazvali ga polje. U `for` petlji pristupamo svakom pojedinom članu preko indeksa koji se nalazi unutar uglatih zagrada.

Gornje polje je jednodimenzionalno. No po volji možemo stvarati višedimenzionalna polja, no potrebno je unaprijed definirati njegove dimenzije (u novijem standardu C99 to više nije potrebno). Primjer za dvodimenzionalno polje veličine 10 x 10

```
int polje[10][10];
polje[0][1] = 5;
```

Dvodimenzionalno polje možemo zamisliti kao tablici u kojoj je prvi indeks redak, a drugi stupac. U gornjem primjeru smo stvorili polje te smo u prvi redak (indeks 0) i drugi stupac (indeks 1) upisali broj 5.

Kada stvorimo novo polje na gore navedeni način mi ne znamo koje su vrijednosti pohranjene u njega. Slično kao i za varijable postoji mogućnost da inicijaliziramo članove polja. Kao primjer stvorimo polje od 5 članova tipa `float` i inicijaliziramo prve tri vrijednosti. To možemo napraviti naredbom

```
float x[5] = {0.4, -0.1, 2};
```

Na ovaj način smo postavili vrijednosti za prva tri člana polja. Ako smo na ovaj način inicijalizirali pojedine članove polja, svi ostali članovi bit će 0. Stoga ako recimo želimo napraviti polje koje će brojati koliko će se puta koji broj od 0 do 99 potrebno je sve njegove elemente postaviti na 0. To ćemo najlakše učiniti s naredbom

```
int brojac[100] = {0};
```

Moramo voditi računa kada na ovaj način inicijaliziramo članova polja da unutar vitičastih zagrada mora biti barem jedan član inicijaliziran te da ne smijemo inicijalizirati više članova od veličine polja. Isto tako ako želimo inicijalizirati veliko polje sa članovima različitim od nula moramo svaki od njih specificirati. Stoga je obično lakše to napraviti naknadno `for` petljom nego da inicijaliziramo član po član.

Primjer za inicijalizaciju polja znakova koji predstavljaju tri osnovne boje (crvenu, plavu i zelenu)

```
char boje[3] = {'C', 'P', 'Z'};
```

Jezik C ostavlja veliku slobodu programeru, ali traži od njega veliku pažnju jer ako mi pokušamo pristupimo članu polje izvan indeksa, bit će nam dopušteno i možemo promijeniti vrijednost nekoj drugoj varijabli.

Niz znakova kao poseban tip podataka se u jeziku C tvori poljem znakova koji završava `null` znakom koji ima vrijednost 0. Ako pišemo `null` znak kao znak u jednostrukim navodnicima onda ga pišemo kao `'\0'`. Moramo voditi računa da osiguramo dovoljan broj članova polja da on uključuje i `null` znak. Na primjer ime Marko možemo smjestiti u polje znakova na ovaj način

```
// Dvostruki navodnici nam označavaju niz znakova.
char ime[6] = "Marko";
// drugi nacin pisanja. Nulu na kraj se dodaje automatski
char ime[6] = {'M', 'a', 'r', 'k', 'o'};
```

U programskom jeziku C++ imamo unaprijed definiran tip podataka `string` za nizove znakova.

3.4.2 Pokazivači

Jezik C za razliku od nekih drugih jezika dopušta nam i pristup preko memorijskih adrese. Jedna od glavnih primjena pokazivača je kada u neku funkciju prenosimo vrijednost. Primjer za to smo već imali sa `scanf` naredbom kada smo prenosili adresu varijabli.

Pokazivači su zaseban tip podataka i svaki od standardnih tipova podataka ima svoj tip pokazivača.

Kao primjer možemo stvoriti pokazivač na int te promijeniti vrijednost lokacije na koju taj pokazivač pokazuje.

```
int a = 5;
int *pa = &a;
*pa = 7;
```

U prvoj naredbi stvorili smo i inicijalizirali varijablu `a` tipa `int` koja sada ima vrijednost 5. U drugoj naredbi smo stvorili pokazivač na `int pa` i dodijelili mu adresu varijable `a`. Ovaj korak je važan je da nismo to napravili lokacija na koju bi pokazivao bi bila neka slučajna i kada bi željeli ju mijenjati, bilo bi nam to omogućeno, no ne bismo znali zapravo koju varijablu mijenjamo. U varijabli `pa` tipa pokazivač na `int` se nalazi zapis točne adrese `a`. U zadnjoj naredbi dereferenciramo varijablu `pa` s operatorom `*` tako da sada više ne radimo s adresom nego vrijednošću adrese na koju pokazivač pokazuje. Nakon te naredbe varijable `a` ima vrijednost 7. Vrlo često zbunjuje operator `*` u drugoj i trećoj naredbi. U drugoj naredbi on zajedno s `int` označava tip podataka tj. `int *` (pokazivač na `int`), a u trećoj radimo dereferenciranje pokazivača.

Posebna vrijednost pokazivača je `NULL` vrijednost. Tu vrijednost stavimo da možemo pri provjeri utvrditi da taj pokazivač ne pokazuje niti na koju aktivnu adresu i da ga treba preusmjeriti. Isto tako `NULL` vrijednost u logičkom izrazu se tretira kao logička laž (0). Ako pokušamo derefencirati pokazivač koji pokazuje na `NULL`, dobit ćemo pri izvođenju programa poruku o pristup krivom segmentu memorije (*engl.* segmentation fault) i program će se automatski srušiti.

Poljima podataka možemo pristupati pokazivačima, no moramo voditi računa o tipovima (npr. polju `int`-ova, možemo pristupiti jedino preko pokazivača na `int`). Jednostavan primjer pokazuje kako možemo koristiti pokazivač za pristup članovima polja.

```
int polje[20];
int * ppolje = polje; // polje je sinonim za &polje[0]
*ppolje = 1;
```

U prvom retku stvorili smo polje od 20 `int`-ova. U drugom stvaramo pokazivač na `int` i dodjeljujemo mu adresu prvog člana polja. U zadnjoj naredbi zapisujemo u prvi član polja vrijednost 1. Ako smo htjeli pristupiti vrijednosti drugog člana polja i promijeniti ju to bi mogli učiniti s naredbom

```
*(ppolje +1) = 2;
```

Upravo zbog ove mogućnosti pokazivača da ga koristimo kod polja i da pokazuje na prvi član polja koristimo indekse polja koji počinju od 0.

Zbog mogućnosti da polju pristupamo kao pokazivaču i obrnuto niz znakova smo mogli pridijeliti pokazivaču na znak

```
char *ime = "Marko";
```

važno je napomenuti da niz znakova predan unutar dvostrukih navodnika je konstanta i kako za razliku od polja ne prepisujemo vrijednosti u polje već imamo samo pokazivač na niz znakova u ovom slučaju niz znakova "Marko" nećemo moći mijenjati.

3.4.3 Dinamička alokacija memorije

Do sada smo za sve varijable koristili takozvano statičko alociranje memorije. No ponekad npr. želimo mijenjati veličinu polja tijekom izvođenja programa. To korištenjem statičke alokacije nije moguće. Zbog toga pribjegavamo dinamičkoj alokaciji memorije i za to su nam nužni pokazivači. Pogledajmo kod za dinamičku alokaciju polja od 50 članova i njegovu realokaciju na 10 članova.

```
#include <stdlib.h>
int main(){
   // Alociramo polje od 50 int-ova
   int *polje = (int *) malloc(50 * sizeof(int));
   // Pridjeljujemo vrijednost trećem članu polja (indeks 2)
   polje[2] = 5;
   // Smanjujemo duljinu polja na 10 int-ova
   polje = (int *) realloc(polje, 10 * sizeof(int));
   // dealociramo polje da ga neki drugi program može koristiti
   free(polje);
   return 0;
}
```

S obzirom ćemo koristiti funkcije za dinamičku alokaciju polja (malloc, realloc, free) onda moram uključiti i standardnu biblioteku `stdlib.h` koja te funkcije koristi. Za razliku od toga statičke alokacije memorije, kada radimo dinamičku alokaciju sami moramo voditi računa o tome koliko nam pojedini tip podataka zauzima prostora. Prevoditelj to neće učiniti za nas. Kada dinamički alociramo memoriju, njoj možemo pristupiti samo preko pokazivača. Funkciji `malloc` moramo reći koliko da memorije alocira. U našem slučaju radi se o polju od 50 `int`-ova stoga je ukupna većina potrebna za alokaciju 50 * veličina jednog `int`-a. Veličinu dobijemo `sizeof` naredbom. S obzirom funkcija `malloc` ne zna unaprijed koji tip podataka će se koristi u memoriju ona vraća pokazivač na `void (void *)`. `void` je zapravo bestipni tip podataka i u ovom slučaju trebamo `void *` pokazivač pretvoriti `cast`-

anjem u `int *` pokazivač. Nakon što smo alocirali memoriju i dodijelili pokazivač na nju, možemo dalje slobodno joj pristupati kao kada se radi o statičkoj alokaciji. Primijetite opet da ovaj put s pokazivačem `int *` možemo postupati kao s poljem, pa tako trećem članu pristupamo preko operatora polja. U slučaju da trebamo alocirati dodatnu memoriju ili zaključimo da nam treba manje polje, možemo napraviti realokaciju s naredbom `realloc`. Naredna `realloc` slično izgleda kao i `malloc` jedino je prvi argument pokazivač na memoriju koju želimo realocirati. Sve ostalo je isto. Kada jednom promijenimo veličinu u daljnjem kodu moramo voditi računa o tome jer možemo pisati ili čitati iz dijelova memorije koji više nisu alocirani za nas. S obzirom da memorija koju dodijelimo dinamički je aktivna tijekom cijelog vremena izvođenja programa ako znamo da ju nećemo više trebati s naredbom `free` ju dealociramo. Moramo paziti da u dealokaciji pokušavamo osloboditi ispravan dio memorije, te da istu već nismo oslobodili. Isto tako trebamo voditi računa da uvijek imamo pokazivač na alocirani dio memorije. Ako promijenimo vrijednost toga pokazivača na neku drugi vrijednost, izgubili smo nepovratnu vezu s tim dijelom memorije i više ga nećemo moći osloboditi.

3.4.4 Polja, pokazivači, reference i dinamička alokacija memorije u jeziku C++

C++ je jezik koji je nastao na C-u i zadržao je sve funkcionalnosti C-a s time da je nadograđen. Osim pokazivača u C++ koristimo i reference koje su razliku od pokazivača zapravo aliasi, odnosno drugo ime za neku varijablu. Jednom kada je referenca inicijalizirana varijablom, možemo koristiti ime varijable ili ime reference na nju za pristup varijabli.

Tri osnovne razlike pokazivača i referenci su slijedeće:

- Ne mogu postojati NULL reference. Referenca može biti povezano jedino na legitiman prostor u memoriji
- Za razliku od pokazivača kod kojih možemo promijeniti da više ne pokazuju na jedan podatak nego na drugi kada je jednom inicijaliziramo referencu i vežemo uz podatak, ne možemo postići da pokazuje na neki drugi.
- Reference moraju biti inicijalizirane čim su stvorene. Pokazivače možemo inicijalizirati bilo kada.

Primjer koda u kojem koristimo referencu

```
int i = 4;
int &ref = i;
ref = 5;
printf("%d", i);
```

Nakon izvođenja ovog koda na zaslonu ćemo dobiti 5. U drugoj liniji smo stvorili novu referencu i inicijalizirali ju na varijablu `i`. Kada smo promijenili vrijednost reference promijenila se vrijednost i varijable `i`. Za reference koristimo isti operator kao za adresu &. Razlika je u tome što s `int &` kažemo da referenca koju stvaramo će se referirati na `int`.

U C++ često koristimo dinamičku alokaciju. Umjesto funkcija `malloc` i `free` u jeziku C++ možemo koristiti `new` i `delete` operatore.

Primjer koda za dinamičku alokaciju jedne varijable tipa `int` polja od 10 `int`-ova i oslobađanje memorije možemo napisati:

```
int *r = new int;
int *p = new int[10];
delete r;
delete [] p;
```

U slučaju kada smo dinamički stvorili jednu varijablu (u ovom slučaju `r`) memoriju kasnije oslobađamo s narednom `delete`. Kod polja moramo naznačiti da se radi o polju stoga je naredba `delete []`. Kada koristimo operatore `new` i `delete` nemamo mogućnost realokacije memorije. Stoga je u tome bolje koristi standardne kontejnere poput `std::vector` koji omogućavaju promjenu veličine.

3.5 Funkcije

Vrlo često moramo različite podatke obraditi istim nizom naredbi. U tom slučaju nema smisla kopirati isti niz naredbi samo za zamjenom imena varijable, nego u tom slučaju radimo zasebne cjeline koje se zovu funkcije i koje se mogu pozivati tijekom izvođenja programa s različitim podatcima (vrijednostima). Na ovaj način smanjujemo ukupnu količinu i čitljivost koda te postižemo njegovu modularnost.

Svaki C/C++ program počinje funkcijom `main`. Iz te funkcije možemo pozivati druge funkcije. Te funkcije mogu biti već postojećim bibliotekama kao što su `stdio.h` ili `stdlib.h` koje smo već koristili ili ih možemo sami napisati.

Primjer za funkciju je računanje sume dva cijela broja broja. Kod u kojem su napisana funkcija i njen poziv iz glavne funkcije mogu izgledati ovako (uključivanje biblioteka nije prikazano).

```
int suma (int a, int b) {
      int zbroj;
      zbroj = a + b;
      // u donjoj liniji koda mijenjamo vrijednosti a i b parametara
      // koje smo predali u funkciju. Ova promjena nema utjecaj na
      // vrijednosti a i b u glavnom programu
      a = b = 2;
      return zbroj;
}
int main() {
      int a = 1;
      int b = 0;
      int zbroj;
      zbroj = suma(a, b); // Poziv funkcije. Rezultat se sprema u zbroj
      printf("Zbroj brojeva %d i %d iznosi: %d", a, b, zbroj);
      return 0;
```

```
}
```

Ako pozivamo neku funkciju iz neke druge funkcije ili glavnog programa, ta funkcija treba biti poznata ranije kroz uključenje u zaglavlje programa ili se nalazi iznad funkcije u kojoj je pozivamo. U ovom primjeru napisali smo ju ispred `main` funkcije. Slično kao i funkcija main svaka funkcija u C/C++ treba imati:

- Ime.
- Tip podataka koji vraća (u ovom slučaju `int`). Tip podataka može biti bilo koji smo do sada spominjali uključujući pokazivače.
- Argumente koji se predaj, a nalaze se unutar uglatih zagrada i odvojeni su zarezom (dva cijela broja `a` i `b`). Moramo naglasiti i njihov tip.
- Vitičaste zagrade unutar kojih upisujemo naredbe koje će se izvršavati u funkciji.
- Povratnu vrijednost (ovdje vraćamo zbroj). Izuzetak su funkcije kojima je tip `void` i tada ništa ne vraćamo. Naredba `return` vraća vrijednost i prekida izvršenje funkcije i predaje izvršenje funkciji koja je pozvala i vraćamo se na liniju koda s kojeg je funkcija pozvana.

Funkcija može po potrebi imati i svoje vlastite lokalne varijable i pozivati neke druge funkcije. Kada pozivamo funkciju moramo predati sve tražene argumente. To mogu biti varijable toga tipa, konstante ili izrazi. Umjesto argumenta može biti i neka druga ili ista funkcija ako to ima smisla.

Kada predajemo argumente u funkciju ne predajemo originalne varijable nego njihove kopije koje se izbrišu nakon izlaska iz funkcije. Na taj način što god radili s tim argumentima unutar funkcije neće imati utjecati na originalnim varijablama.

3.5.1 Deklaracija i definicija funkcije

Deklaracija funkcije

Deklaracija funkcije se sastoji od povratnog tipa, imena funkcije i popisa argumenata s navedenim tipom podataka. Ako imamo samo deklaraciju bez tijela funkcije onda ju završavamo s točka zarezom. Primjer deklaracije funkcije je

```
int suma(int a, int b);
```

Deklaracija funkcije je značajna jer nam govori o tome kakve argumente prima određena funkcija. Ta informacije je nužna da bi prevoditelj mogao znati koliko mjesta treba rezervirati pri izradi strojnog koda za argumente. Zbog toga čak nije nužno niti navesti imena argumenata već samo tipove podataka. Za prevođenje dovoljno imati deklaraciju svih funkcija koje se koriste i koje se nalaze napisane iznad mjesta na kojem se prvi put pozivaju. Deklaracija se ne piše unutar drugih funkcija. Drugi način je da spremimo deklaracije u `.h` datoteku koju onda moramo uključiti u kod sa pretprocesorskom naredbom `#include`. Te datoteke nazivamo datotekama zaglavlja. U slučaju kada koristimo standardne funkcije već instalirane na računalu onda se te datoteke nalaze u oštrim zagradama kao što je npr. u ovoj liniji

```
#include <stdio.h>
```

Kada sami kreiramo datoteku zaglavlja, onda moramo navesti konkretan put do nje i takve datoteke stavljamo u navodnike kao što je prikazano u ovoj liniji

```
#include "zaglavlja/moja.h"
```

Ova linija nam kaže da se zaglavlja nalaze u direktoriju zaglavlja u datoteci `moja.h`. Ova notacija vrijedi za operativne sustave bazirane na unixu. Za Windows operacijske sustave koriste se obrnute kose crte.

Deklaraciju funkcije je nužno navesti u bilo kojoj datoteci u kojoj pišemo kod gdje se ta funkcija poziva.

Definicija funkcije

U definiciji funkcije nalazi se i njena deklaracija i tijelo funkcije koje se nalazi unutar vitičastih zagrada. Definicija funkcije mora oblikom u potpunosti odgovarati deklaraciji (tip funkcije, broj redoslijed i tip argumenta); mogu se razlikovati imena argumenata. Svi pozivi funkcije će se prilikom povezivanja programa usmjeriti na tu definiciju.

U jednostavnim programima definicija funkcije može biti unutar iste datoteke s deklaracijom na vrhu, a u složenim programima se obično smješta u posebnu datoteku s deklaracijama u `.h` datoteci.

3.5.2 Lista argumenta

Argumenti koji se predaju funkciji na obradu i koji se koriste kao lokalne varijable unutar funkcije. Kako se radi o kopijama originalnih podataka sve izmjene unutar funkcije neće imati nikakav utjecaj na originalne podatke. Po izlasku iz funkcije sve lokalne varijable i svi predani argumenti se brišu.

Ako želimo mijenjati varijable iz funkcije koja je pozvala određenu funkciju onda trebamo predati pokazivače ili reference (u C++) na te varijable. Na taj način preko pokazivača ili referenci imamo pristup varijablama iz funkcije koja je pozvala tu funkciju. Sam pokazivač ili referenca nakon završetka funkcije će biti obrisani, no ako se preko njih mijenjala vrijednost neke varijable ta promjena će ostati. Na osnovu toga da li predajemo same varijable ili njihove pokazivače ili reference razlikujemo dva tipa prijenosa argumenata:

- Prijenos po vrijednosti.
- Prijenos po referenci.

Ako želimo da se vrijednost nekoga argumenta ne mijenja u funkciji (prije svega je važno za pokazivače i reference) upotrebljavamo kod navođenja argumenata da je taj argument `const` čime se njegova vrijednost ne mijenja.

Ukoliko želimo predati polje u funkciju dovoljno je predati pokazivač na prvi član polja. Isto tako kako nemamo informaciju o dimenziji polja trebamo predati i tu vrijednost. Specifičan slučaj je niz znakova kada znamo da je zadnji član polja `null` znak. U tome slučaju nam nije nužno slati i broj članova polja.

Dok u C++ jeziku možemo imati argumente s predefiniranom vrijednošću to u C-u nije moguće. Primjer za C++ je funkcija za računanje sume brojeva

```
int sum(int x, int y, int z = 0, int w=0) {
      return (x +y + z + w);
}
```

Argumenti s predefiniranom vrijednošću trebaju doći na kraju funkcije. Čim jedan argument ima predefiniranu vrijednost to moraju imati i svi oni iza njega. Dovoljno je da argument s predefiniranom vrijednošću navedemo samo u deklaraciji.

Funkcija `printf` je dobar primjer funkcije koja ima neodređeni broj parametara. Takve funkcije na kraju liste argumenata imaju tri točke (…). Obično se kod takvih funkcija ukupan broj argumenata može izvući iz prenesenih argumenata. Kod `printf` ili `scanf` funkcije je to niz znakova s modifikatorima.

3.5.3 Definicija i deklaracija varijabli

Deklaracija varijable opisuje ime i tip, a definicija se mora nalaziti negdje drugdje. Definicija varijable opisuje ime, tip, rezervira memoriju i eventualno inicijalizira vrijednost te ujedno služi i kao deklaracija.

Općenito kada dodajemo novu varijablu pišemo

```
smjestajni_razred tip_podataka varijabla …;
```

Smještajni razred i mjesto na kojem ćemo definirati varijablu određuju postojanost i važenje varijable u memoriji. Postojanost ili trajnost varijable određuje područje programskog koda tijekom čijeg izvršavanja sadržaj varijable ostaje sačuvan. Područje ili doseg važenja varijable određuje područje programskog koda unutar kojega se varijabla može koristiti.

Četiri osnovna smještajna razreda su: `auto, register, static` i `extern`.

Smještajni razred auto predstavljaju lokalne varijable i one vrijede od mjesta definicije do kraja funkcije ili bloka (unutar vitičastih zagrada) unutar kojeg su definirane. Sve varijable koje nemaju naznačen razred smatraju da imaju razred `auto`.

Smještajni razred `register` predstavlja preporuku prevodiocu da pohrani vrijednost varijable u memoriju koja se nalazi u procesorima i koju nazivamo registrima. Na ovaj način će procesor jako brzo moći doći do tih podataka. Smještanje varijabli u registre se koristi kada želimo optimirati naš kod da smanjimo vrijeme izvođenja. Za rad s registrima postoje posebni skupovi instrukcija poput SSE4, AVX i AVX2. Isto tako rad s registrima se koristi kada programiramo na grafičkim karticama. Varijable koje imaju ovaj razred je moguće, kao i razred `auto`, jedino definirati unutar bloka naredbi ili unutar funkcije.

Smještajni razred `static` se može definirati unutar funkcije ili bloka ili izvan njih. Ako se definira unutar funkcije ili bloka onda će varijabla važiti do njihovog kraja, a ako je izvan

onda će važiti od mjesta definicije do kraja modula ili u modulu u kojem je definirana (ovisno od prevodioca). Ono što ju čini različitom od prethodnih razreda je da vrijedi od početka do kraja programa. Ako se eksplicitno ne inicijalizira tijekom definicije njena vrijednost se postavlja na 0. Primjer korištenja razreda `static` je korištenja `static` varijable unutar funkcije za brojanje poziva funkcije.

Smještajni razred `extern` se koristi za deklaraciju varijabli. Često varijable ovoga razreda nazivamo globalnim varijablama. Za razliku od ostalih razreda varijabla će biti definirana negdje drugdje. Slično kao i kod razreda `static` ova varijabla će varijabla važiti do njihovog kraja funkcije ili bloka u kojem je deklarirana, a ako je izvan onda će važiti od mjesta deklaracije do kraja modula. Isto tako varijabla traje od početka do kraja projekta te ako nije eksplicitno inicijalizirana tijekom definicije vrijednost joj se postavlja na 0.

3.5.4 Preopterećenje funkcija

Preopterećenje funkcije (*engl.* overloading) je koncept programiranja u kome se dozvoljava programerima da definiraju dvije ili više funkcija istoga imena i u istom području važenja. Te funkcije imaju isto ime, ali imaju različiti broj argumenata ili su onih različitog tipa. Preopterećenje funkcije nije moguće u jeziku C, ali se u jeziku C++ često koristi.

Primjer preopterećenja bez navedeni naredbi preprocesoru

```
void prikazi(int);
void prikazi(float);
void prikazi(int, float);

int main() {
    int a = 5;
    float b = 5.5;
    prikazi(a);
    prikazi(b);
```

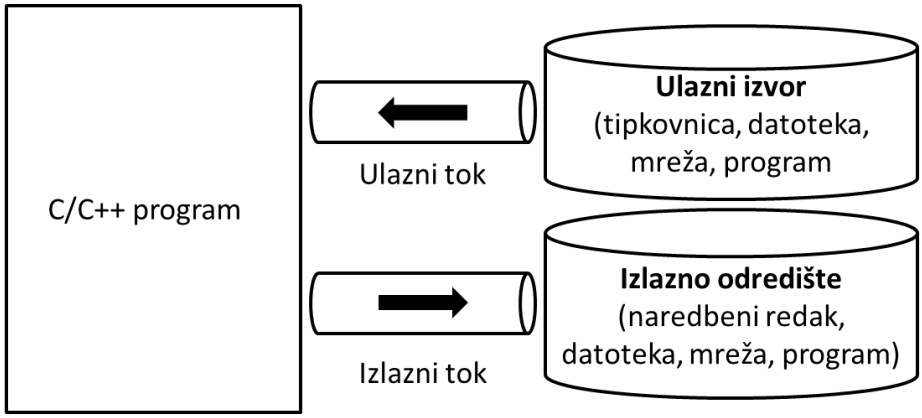
```
    prikazi(a, b);
    return 0;
}

void prikazi(int var) {
    printf("Cijeli broj: %d\n", var);

void prikazi(float var) {
    printf("Realni broj: %f\n", var);
}

void prikazi(int var1, float var2) {
    printf("Cijeli broj: %d\n", var1);
    printf("Realni broj: %f\n", var2);
}
```

Kada pokrenemo kod funkcija prikazi će se pokrenuti tri puta i svaki put ispisat različite poruke s različitim vrijednostima. Prvi put ispisati će se cijeli broj, drugi put realni, a treći put oba.

3.6 Ulazno izlazni tokovi

Važan dio funkcioniranja svakoga programa je komunikacija s ulaznim i izlaznim uređajima. Da bismo mogli ostvariti komunikaciju s tim uređajima u jezicima C i C++ koristimo ulazne i izlazne tokove (engl. *input and output streams*). Tokovi predstavljaju tok okteta iz i u program baš kao što voda teče kroz cijevi. Ovi tokovi podataka su realizirani tako da je korisniku svejedno s kojim ulaznim ili izlaznim uređajem komunicira i nema potrebu direktnog upravljanja s tim uređajima.

Slika 3.1 prikazuje neke od najčešćih izlaza i ulaza u program povezanih s tokovima podataka.



Ulazni formati podataka:

- Tekst: char, wchar_t
- int, float, double

Izlazni formati podataka:

- Tekst u različitim ekodinzima (ASCII, UTF-8, UTF-16, ISO-8859-2, itd.)
- Binarno (sirovi okteti)

Slika 3.1 Primjer ulaznih i izlaznih tokova podataka

Pri pokretanju svakoga programa operacijski sustav tri standardna toka podataka:

- Standardni ulaz (`stdin`).
- Standardni izlaz (`stdout`).
- Standardna pogreška (`stderr`).

`Stdin` načešće predstavlja tipkovnicu, `stdout` zaslon računala, a preko `stderr` program javlja pogreške i razne dijagnostike pokrenute od strane programa.

Do sada upoznali naredbe `printf` i `scanf` koje služe za ispis podataka na zaslonu i unos podataka preko tipkovnice koji služe kao standardni ulaz i izlaz. Postoji još čitav niz funkcija za unos preko `stdin` i ispis preko `stdout`. Deklaracije ovih funkcije se u jeziku C nalaze u biblioteci `stdio.h`, a u jeziku C++ u biblioteci `iostream`.

Osim standardnih tokova podatka korisnik može otvoriti tokove podataka prema drugim odredištima kao što su datoteke koje mogu biti i izvor i odredište pojedinih tokova podataka.

Općenito da bismo napravili prebacili podatke u ili iz programa potrebno je napraviti slijedeće:

1. Stvoriti tok podataka (objekt ili struktura zavisno o jeziku).
2. Povezati stvoreni tok s ulazno-izlaznim uređajem s kojim želimo ostvariti prijenos podataka.
3. Obaviti ulazno/izlazne operacije preko funkcija koje predstavljaju sučelje toka podataka. Neke funkcije pretvaraju podatke između vanjskog i unutarnjeg formata (formatirane ulazno/izlazne operacije), dok druge to ne čine (binarne ulazno/izlazne operacije).
4. Odspojiti tok podataka od ulazno/izlaznog uređaja (npr. zatvoriti datoteku).
5. Osloboditi memoriju koju je zauzeo tok podataka.

Za razumijevanje tokova podataka važno je razumjeti kako se podaci šalju i spremaju na odredište. U tome procesu koristimo međuspremnike (engl. *buffer*). Međuspremnik je blok memorije koji pripada toku i koristi za privremeno zadržavanje podataka. Ako želimo podatke prebaciti u datoteku, oni će prvo biti spremljeni u međuspremnik i tamo će se nalaziti dok ih se ne prenese u datoteku u dijelovima. Slično kod ulaznih tokova podataka, ulaz dobivamo u blokovima, a ne znak po znak. Zbog toga postižemo veću efikasnost jer su ulazno/izlazne jedinice puno sporije od operacija s memorijom.

Razlikujemo tri vrste međuspremanja podataka:

- bez međuspremanja,
- međuspremanje bloka,
- međuspremanje retka.

Osnovni način rada je kada ne koristimo međuspremanje već samo zapisujemo npr. tekst u datoteku kako se oni pojavljuju. Kod međuspremanja bloka podaci se spremaju u međuspremnik i spremaju ili čitaju iz njega kao blok. Kod međuspremanja retka znakovi se spremaju dok ne dođe znak za prelazak u novi red. Standardni tokovi `stdin` i `stdout` koriste međuspremanje bloka kada nisu usmjereni na interaktivni uređaj, inače koriste

međuspremanje linije. `stderr` tok se šalje bez međuspremanja. Vrlo je važno biti svjestan pri korištenju funkcija za rad sa standardnim tokova kako rade jer često se zna dogoditi da podatci kod čitanja ostanu u međuspremniku pa možemo dobiti pogrešan rezultat.

3.7 Tokovi podataka u jeziku C

3.7.1 Standardni tokovima podataka

Osnovne funkcije za standardni izlazni tok podataka `stdout` u programskom jeziku C su:

- `printf()`
- `putchar()`
- `puts()`

Funkciju `printf()` smo obradili u ranijim poglavljima.

Funkcija `putchar` ispisuje jedan znak na zaslon. Njena deklaracija izgleda ovako:

```
int putchar(int char);
```

Funkcija `puts` ispisuje niz znakova na ekran. Njena deklaracija izgleda ovako:

```
int puts(const char * str);
```

Kod standardnog ulaznog toga podataka imamo slične ekvivalentne funkcije:

- `scanf()`
- `gethcar()`
- `gets()`

Funkciju `scanf()` smo ranije obradili.

Funkcija `getchar()` uzima jedan znak sa standardnog ulaza-tipkovnice i vraća ga pretvorenog u tip `int`. Tu funkciju smo koristili na početku kada smo željeli da zaustavimo izvršavanje programa dok ne unesemo jedan znak. Na taj način smo pratili da li program ispravno na ispisuje na zaslon. Njena deklaracija je:

```
int getchar(void);
```

Funkcija `gets` čita sve znakove sa standardnog ulaza do znaka za kraj retka i vraća pokazivač na mjesto u memoriji gdje se taj niz nalazi. Njena deklaracija je:

```
char * gets ( char * str );
```

3.7.2 Rad s vlastitim tokovima podataka

Osim standardnih tokova podataka, korisnik može kreirati vlastite tokove podataka. Najčešće to čini za spremanje podataka u datoteku ili čitanje podataka iz nje. U daljem tekstu ćemo se fokusirati isključivo na rad s datotekama.

Postoje dva načina na koja možemo podatke spremati u datoteku:

- Tekstualni zapis,
- Binarni zapis.

Kod tekstualnog zapisa zapisujemo nizove znakove gdje retci završavaju zapisom kraja retka. Svaki znak neovisno da li bio znak ili broj zapisujemo u ASCII formatu te koristimo formatiranje slično kao u `printf` funkciji.

Kod binarnog direktno prepisujemo sadržaj memorije pa se znakovi zapisuju u ASCII formatu dok kod brojeva je važan tip podataka. Npr. ako imamo broj 1 zapisan u 32-bitnom `int` zapisu, svih 32 bita bit će prepisano u datoteku.

Zbog toga što kod spremanja tekstualnih datoteka koristimo formate zapisa te datoteke nazivamo i formatiramo, a binarne neformatirane.

Vrlo je važno znati na koji su podaci zapisani u datoteku da ne napravimo grešku pri čitanju. Zavisno o tipu zapisa koristimo i različite funkcije.

Prvi korak rada s datotekama je stvaranje toka podataka. U jeziku C taj tok podataka predstavlja struktura tipa `FILE` definirana u `stdio.h` biblioteci. Ova struktura sadrži različite informacije o datoteci kao što je njezina veličina. Za rad sa strukturom koristit ćemo pokazivač na nju.

Za otvaranje datoteke i njeno pridruživanje toku podataka koristimo funkciju `fopen()` koja se također nalazi u biblioteci `stdio.h`. Njezina deklaracija izgleda ovako:

```
FILE *fopen(char *ime_datoteke, char *mod);
```

`ime_datoteke` je ime datoteke koju želimo otvoriti ili stvoriti i predajemo ga kao niz znakova. Niz znakova `mod` određuje način pristupa datoteci (Tablica 3.7). Važno je pažljivo

odabrati mod rada jer ako odabiremo modove koji počinju s `w` postojeće datoteke bit će obrisane. Isto tako potrebno je znati o kojem tipu datoteka se radi te po potrebi koristiti oznaku `b` za binarni tip datoteke.

Tablica 3.7 Načini rada s datotekom

Mod	**Značenje**
r	Otvaranje tekstualne datoteke za čitanje.
w	Kreiranje tekstualne datoteke za upis.
a	Dodavanje teksta u postojeću tekstualnu datoteku.
rb	Otvaranje binarne datoteke za čitanje.
wb	Otvaranje binarne datoteke za upis.
ab	Dodavanje u postojeću binarnu datoteku.
r+	Otvaranje tekstualne datoteke za čitanje/upis.
w+	Kreiranje tekstualne datoteke za čitanje/upis.
a+	Dodavanje ili stvaranje tekstualne datoteke za čitanje/upis.
r+b	Otvaranje binarne datoteke za čitanje/upis.
w+b	Kreiranje binarne datoteke za čitanje/upis.
a+b	Dodavanje u binarnu datoteku za čitanje/upis.

Funkcija `fopen` vraća pokazivač na `FILE` strukturu. U slučaju da funkcija ne uspije, vraća se NULL pokazivač. Stoga je uvijek nakon `fopen` funkcije potrebno provjeriti da li je proces otvaranje datoteke prošlo uspješno.

Primjer koda s otvaranjem datoteke za čitanje iz nje i provjerom povratne vrijednosti funkcije `fopen`

```
FILE *fp;
if((fp = fopen("moja_datoteka", "r")) == NULL) {
      printf("Pogreška u otvaranju datoteke !!\n");
```

```
exit(1);
}
```

Nakon što smo odradili sve što smo planirali s datotekom istu je potrebno zatvoriti. Za to koristimo funkciju `fclose()` čija deklaracija je:

```
int fclose(FILE *fp);
```

Funkcije `fclose()` zatvara datoteku pridruženu pokazivaču `fp` koji mora biti ispravan pokazivač koji pokazuje na vrijednost dobivenu funkcijom `fopen()`. Istovremeno se prekida i tok prema toj datoteci. Uspješna funkcija `fclose()` vraća 0, a ako je došlo do pogreške vraća se EOF (engl. *end of file*).

Funkcije za rad s tekstualnim datotekama

Funkcije koje koristimo pri radu s tekstualnim datotekama su slične onima za učitavanje znakova s tipkovnice i njihov prikaz na zaslonu. Njihova imena su dobivena dodavanjem slova na f na početku.

Prve dvije funkcije za rad s tekstualnim datotekama su `fputs()` i `fgets()`. Njihove deklaracije su slijedeće:

```
int fputs(char *niz, FILE *fp);
char *fgets(char *niz, int broj_znakova, FILE *fp);
```

Funkcija `fputs()` zapisuje niz znakova na koje pokazuje pokazivač `niz` u file pridružen pokazivaču `fp`. Funkcija vraća nenegativnu vrijednost ako je zapisivanje uspjelo, a EOF ako nije. Null znak kojim označavamo kraj niza znakova se zapisuje, a automatski se dodaje oznaka za kraj retka.

Funkcija `fgets()` čita znakove iz datoteke pridružene pokazivaču `fp` i prepisuje ih u memorjii u niz znakova na koje pokazuje pokazivač `niz` sve dok se ne prepiše `broj_znakova -1` znak, ne dođemo do znaka kraj retka ili kraj datoteke. Ukoliko smo pročitali i znak za kraj retka njega također prepisujemo, a na kraj prepisanoga niza znakova dodaje se null znak čime označavamo da se radi o kraju niza znakova. Važno je sjetiti se da smo prepisali i znak za kraj retka pa ukoliko ga ne trebamo, trebamo ga ukloniti. Funkcija u

slučaju ispravnog izvršenja vraća pokazivač `niz`, a ako je došlo do pogreške NULL pokazivač.

Funkcija `gets()` s kojom unosimo tekst preko tipkovnice ima nedostatak jer ne znamo koliko je zapravo velik tekst koji ćemo unijeti te ako nismo rezervirali dovoljno prostora za taj tekst može nam se dogoditi da prebrišemo vrijednosti nekih drugih varijabli što predstavlja i sigurnosni propust. Stoga se preporuča umjesto nje korištenje funkcije `fgets()` koja omogućava da ograničimo broj znakova koje unosimo. Kao pokazivač na tok podataka samo napišemo ključnu riječ `stdin` što predstavlja standardni tok za unos podataka. Slično vrijedi i za sve ostale funkcije za unos teksta iz standardnog toka.

Funkcije za rad s datotekama koje odgovaraju funkcijama `printf()` i `scanf()` su `fprintf()` i `fscanf()` koje koristimo za rad s tekstualnim datotekama.

Deklaracije funkcija `fprintf()` i `fscanf()` su:

```
int fprintf(FILE *fp, char *control-string, ...);
int fscanf(FILE *fp, char *control-string ...);
```

Za razliku od klasičnih funkcija razlika je da u ovom slučaju imamo pokazivač na tok, sve ostalo je isto. Za razliku od `fputs()` i `fgets()` ove dvije funkcije omogućavaju preciznije formatiranje.

Funkcije za rad s binarnim datotekama

Kod rada s binarnim datotekama koristimo novu klasu funkcija koje do sada nismo sretali. To su funkcije koje direktno prenose oktete binarnih podataka iz memorije u datoteku. Dvije osnovne funkcije za rad s binarnim datotekama su fread() i fwrite() s kojima zapisujemo i čitamo. Njihove deklaracije su:

```
size_t fread(void *buffer, size_t velicina, size_t n, FILE *fp);
size_t fwrite(void *buffer, size_t velicina, size_t n, FILE *fp);
```

Funkcija `fread()` čita direktno podatke s diska i kopira ih u memoriju s time da ne zna ništa o tipu podataka koji čita, već mi moramo te podatke interpretirati. Ona formalno čita iz datoteke pridružene pokazivaču `fp`, `n` objekata od koji je svaki `velicina` okteta dugačak te ih sprema na memorijsku lokaciju na koju pokazuje pokazivač `buffer`. Kao povratnu

vrijednost vraća broj pročitanih objekata. Ako je povratna vrijednost 0, niti jedan objekt nije pročitan, te se ili dogodila pogreška ili se pri čitanju došlo do kraja datoteke.

Funkcija `fwrite()` zapisuje sadržaj na način da zapisuje u datoteku kojoj je pridružen pokazivač `fp`, `n` objekata, od kojih je svaki `velicina` okteta dugačak iz lokacije na koju pokazuje `buffer`. Funkcija vraća broj uspješno upisanih. Ova vrijednost je manja od `n` jedino ako se dogodi greška pri pisanju.

Može se primijetiti da je kod obje funkcije `buffer` pokazivač na `void`. Razlog za to je što se želi omogućiti da se bilo koji tip podataka može pročitati i zapisati, a lako možemo naredbom `cast` promijeniti tip podataka. Najčešće zapisujemo složene tipove podataka, odnosno C strukture.

Primjer korištenja `fread()` i `fwrite()` funkcija u kome u datoteku zapisujemo vrijednost varijable i (100) tipa `int`:

```
#include <stdio.h> /* datoteka zaglavlja */
#include <stdlib.h> /* datoteka zaglava u kojoj je funkcija exit */

int main(void){
      FILE *fp; /* pokazivač na strukturu FILE */
      int i;
      /* otvaranje datoteke za unos */
      if ((fp = fopen("moja_datoteka", "w"))==NULL){
            printf("Ne mogu otvoriti datoteku\n");
            exit(1);
      }
      i=100;
      if (fwrite(&i, 2, 1, fp) !=1){
            printf("Greška pri upisu!\n");
            exit(1);
```

```
        }
        fclose(fp);


        /* otvaranje datoteke za čitanje */
        if ((fp =fopen("moja_datoteka", "r"))==NULL){
              printf("Greška pri čitanju datoteke");
              exit(1);
        }
        printf("i je %d",i);
        fclose(fp);
}
```

Funkcije za kretanje po datotekama i rad s datotečnim sustavom

Pri čitanju i pisanju u tekstualne datoteka, uvijek počinjemo od početka datoteke, a nastavljamo nakon zadnjeg dijela kojeg smo pročitali ili zapisali. Trenutna pozicija od koje ćemo nastaviti pisanje ili čitanje je odnosno pokazivač pozicije u datoteci je spremljen u `FILE` strukturi. Često međutim želimo početi čitati ili pisati od točno određenog mjesta u datoteci. Za određivanje mjesta gdje se nalazimo u datoteci te kretanje po njoj služe slijedeće funkcije: `ftell(), fseek()` i `rewind()` koje rade s `FILE` strukturom.

Funkcija `ftell()` nam vraća trenutni položaj u datoteci u broju okteta u odnosu na njen početak. Njezina deklaracija je:

```
long int ftell(FILE *fp);
```

Funkcija vraća trenutnu vrijednost ili indikator pozicije. Ako se dogodi pogreška vraća -1L, a globalna varijabla `errno` se postavlja na neku pozitivnu vrijednost.

Funkcija `fseek()` na omogućava kretanje po datoteci. Njena deklaracija je:

```
int fseek(FILE *fp, long pomak, int labela);
```

Osim pokazivača na tok podataka, imamo varijablu `pomak` koja se odnosi na pomak u oktetima u odnosu na `labela`.

`labela` može imati tri vrijednosti:

- `SEEK_SET` – početak datoteke,
- `SEEK_CUR` – trenutna pozicija,
- `SEEK_END` – kraj datoteke.

Funkcija `fseek()` pomiče trenutnu poziciju u datoteci, vraća 0 ako je pomak uspio ili broj različit od 0 ako nije.

Primjer korištenja naredbi `ftell()` i `fseek()` možemo vidjeti ovdje:

```
#include <stdio.h>
#include <stdlib.h>

int main (void) {
      FILE *fp;
      int len;
      fp = fopen("moja_datoteka", "r");
      if(fp == NULL) {
            printf("Greška pri otvaranju datoteke\n!");
            exit(1);
      }
      fseek(fp, 0, SEEK_END); /*Odlazak na kraj datoteke*/
      duljina = ftell(fp);
      fclose(fp);
      printf("Ukupna duljina moja_datoteka = %d okteta\n", duljina);
      return(0);
}
```

Ovaj programski odsječak ispisuje duljinu datoteke `moja_datoteka`. Da bi to izveo prvo je otvorio datoteku za čitanje, a nakon toga je smjestio pokazivač na kraj datoteke i pozivom funkcije `ftell()` odredio duljinu u oktetima.

Kada se želimo pozicionirati na početak datoteke to možemo učiniti s naredbom `fseek(fp, 0, SEEK_SET)`. Umjesto toga mogli smo koristiti i funkciju `rewind()` koja ima istu svrhu, a čija deklaracija glasi:

```
void rewind(FILE *fp);
```

Osim funkcija za rad s datotekama postoje i funkcije koje rade s datotečnim sustavom. Ovdje navodimo samo funkciju `remove()` s kojom brišemo određenu datoteku. Njena deklaracija je:

```
int remove(char *ime_datoteke);
```

3.7.3 Tokovi podataka u jeziku C++

U ovom poglavlju ćemo dati kratak pregled tokova u jeziku C++. Za razliku od jezika C ovdje se tokovi spremaju u objekte.

Standardni tokovi

Slično kao kod jezika C postoje četiri standardna toka:

- `cin` – ulazni tok,
- `cout` – izlazni tok,
- `cerr` – tok pogreške,
- `clog` – tok logiranja.

Tokovi `cerr` i `clog` su isto izlazni tokovi. Razlikuju se od `cout` toka time da su namijenjeni za specijalne svrhe: prikaz pogrešaka i logiranje. Iako u većini okružja oni se ispisuju na zaslon moguće ih preusmjeriti.

Standardni izlazni tok se koristi zajedno s operator umetanja << . Ispod su primjeri za ispisivanje različitih vrijednosti

```
cout << "izlazni tekst!"; // ispisujemo rečenicu na zaslon računala
cout << 10; // ispisujemo konstantu 10 na zaslon
cout << i; // ispisujemo vrijednost varijable i;
```

Korištenjem nekoliko operatora umetanja dobijemo ulančavanje. Na primjer možemo napisati:

```
cout << "Broj " << x << "je veći od broja " << y;
```

Ako želimo nakon ispisivanja preći u novi redak možemo koristi ili znak za prelazak u novi red `\n` ili `endl` manipulator. Kao primjer naredbe:

```
cout << "ovo je prvi redak.\n";
cout << "ovo je drugi redak." << endl;
```

će ispisati

```
Ovo je prvi redak.
Ovo je drugi redak.
```

Standardni ulaz je za mnoge programe tipkovnica, a u jeziku C++ objekt toka preko kojeg pristupamo tipkovnici je `cin`.

Slično kao i kod izlaznog toka `cout` koristimo operator za izvlačenje >>.

U slijedećem kodu:

```
int broj_bodova;
cin >> broj_bodova;
```

u prvom retku deklariramo varijablu tipa `int` u koju spremamo broj bodova na testu. Druga linija izvlači sa standardnog ulaza vrijednost. Program će stati sve dok ne unesemo traženi podataka. Nakon toga će spremiti unesenu vrijednost.

Slično kao i kod standardnog ulaznog toka, možemo ulančavati zahtjeve za unos. Tako na primjer:

```
cin >> prvi_broj >> drugi_broj;
```

je ekvivalentno

```
cin >> prvi_broj;
cin >> drugi_broj;
```

U slučaju kada želimo unijeti niz znakova moramo voditi računa da će se kod izvlačenja bjeline (razmaci, tabulatori, novi reci) biti tretirani kao vrijednosti koji prekidaju izvlačenje.

Ukoliko želimo cijelu liniju teksta potrebno je koristiti getline() funkciju koja čija deklaracija su:

```
istream& getline(istream& is, string& str, char delim);
istream& getline(istream& is, string& str);
```

pri čemu je `is` objekt klase `istream` i kaže funkciju o toku s kojeg se čita ulaz (obično je to cin), `str` je objekt `string` (niz znakova u C++) i ulaz se sprema u taj objekt nakon čitanja iz toka, te `delim` terminalni znak koji kaže funkciji da zaustavi čitanje nakon što dohvati taj znak. U drugoj deklaraciji je sve slično prvoj, uz razliku da se pretpostavlja da je terminalni znak `\n`.

Primjer jednostavnog C++ koda s korištenjem funkcije getline():

```
#include <iostream>
#include <string>
using namespace std;

int main (){
      string str;
      cout << "Unesite vaše ime: \n";
      getline (cin, str);
      cout << "Pozdrav, " << str << " dobro došli na Informatiku !\n";
      return 0;
}
```

4 Algoritmi i programi

Algoritam je precizno opisan način rješavanja zadanog problema koji jednoznačno određuje što treba napraviti. Na ulazu algoritma imamo početne objekte koji pripadaju nekoj klasi objekata na kojima se obavljaju operacije. Kao ishod algoritma imamo završne objekte. Algoritmi moraju imati konačan broj koraka pri čemu je svaki korak opisan instrukcijom.

Za računalne algoritme važno je da se rezultat može dobiti u konačnom vremenu. Važni pojmovi kod razmatranja algoritama su: djelotvornost i učinkovitost. Da bi algoritam bio djelotvoran potrebno je moći u konačnom vremenu dobiti točan rezultat koristeći olovku i papir. Učinkovitost se mjeri u odnosu na utrošene resurse kao što su: trajanje odnosno broj ciklusa procesora, količina radne memorije i diska. Postoje algoritmi koji su djelotvorni ali nisu učinkoviti kao što svođenje množenja na ponavljanje zbrajanja.

Za uspješno programiranje važne su i strukture podataka. Strukture podataka opisuju na koji način su podaci spremljeni u memoriju i kakva je veza među njima. Do sada smo radili linearnu strukture podataka polje. Postoje brojne druge strukture podataka pri čemu su često koriste strukture u obliku stabala jer se brzo mogu pretraživati.

Program se sastoji od niza algoritama i struktura podataka opisani u nekom programskom jeziku, a jednoznačno određuje što računalo treba napraviti.

Da bi se naučilo programirati potrebno savladati sintaksu nekog programskog jezika i steći osnova intuitivna znanja glede algoritmizacije problema opisanog riječima.

4.1.1 Analiza složenosti algoritama

Za svaki problem koji rješavamo algoritamski postoji više od jednog mogućeg rješenja pri čemu težimo najdjelotvornijem rješenju. Postavlja se pitanja kako odrediti koji je algoritam djelotvorniji. Za to nam služi metoda analize složenosti algoritma.

Analizu možemo provesti a priori ili a posteriori. U a priori analizi procjenjujemo trajanje programa na osnovu broj operacija odnosno utrošak memorije. A priori analiza zanemarujemo utjecaj računala, programskog jezika i prevoditelja. U a posteriori analizi za različite skupove podataka mjerimo vrijeme izvođenja i količinu memorije potrebnu za

izvođenje algoritma. A posteriori analizu je jednostavno izvršiti ako imamo sve relevantne testne podatke kojim ćemo pokriti što više mogućih slučaja.

A priori analiza

Da bi smo ravnopravno usporedili algoritme moramo ispuniti slijedeće pretpostavke:

- Imamo sekvencijalno jednoprocesorsko računalo.
- Vrijeme dohvata sadržaja iz memorijske lokacije je fiksno.
- Vrijeme obavljanja operacija (aritmetičke, logičke, pridruživanje, poziv funkcije) je ograničeno nekom konstantom kao gornjom granicom.

Uzevši u obzir gore navedene pretpostavke za mjerenje trajanja računamo broj potrebnih operacija za izvršenje algoritma. Kada radimo analizu radimo ju u najboljem, najgorem i prosječnom slučaju s time da je naglasak prvenstveno na zadnja dva. Nema smisla proglasiti najboljim algoritmom onaj koji postiže najbolje rezultate u samo jednom slučaju, a za sve ostale slučajeve je lošiji.

Kada radimo procjenu složenosti obično se vežemo uz veličinu podataka. Npr. ako imamo na ulazu polje od `n` članova, složenost ćemo izraziti funkcijom od n.

U pravilu nas zanimaju tri mjere kojima se uspoređuje efikasnost algoritama:

- Najgori mogući slučaj.
- Prosječan slučaj, odnosno aritmetička srednja vrijednost svih slučaja.
- Brzina izvršavanja na unaprijed određenom skupu podataka.

U nekim slučajevima najčešća vrijednost zna biti veća od prosječne. Tada će prosječan slučaj podcijeniti vrijeme/resurse potrebe za dani ulaz. Isto tako ako ulaz nije slučajan, nego se neke vrijednosti češće ponavljaju, stvarne performanse algoritma bit će lošije nego u prosječnom slučaju. Za razliku od prosječnog slučaja **amortizirana analiza** uzima u obzir ukupne performanse niza operacija umjesto samo jedne.

Kao primjer računanja složenosti možemo uzeti u obzir dinamičko umetanja članova polja. Algoritam radi tako da alocira fiksnu veličinu polja na početku. Nakon toga se dodaju članovi polja. U slučaju kada nam trenutna veličina polja nije dovoljna, alociramo dva puta veće polje od staroga. U najgorem slučaju ubacivanje člana polja zahtijeva *n* operacija.

Međutim ako možemo garantirati da je taj najgori slučaj jako rijedak, u amortiziranoj operaciji potreban je samo konstantan broj operacija nevezan za *n*.

Ponekad je zanimljivo vidjeti kako se algoritam ponaša za najbolji slučaj, no ta mjera nam ne daje pravu sliku efikasnosti algoritma.

Način na koji računamo složenost najbolje je proći kroz primjere. Tablica 4.1 prikazuje primjere izračuna broja operacija. Po definiciji kada imamo samo jednu liniju koda brojimo ju kao 1. U slučaju kada imamo petlje moramo izbrojati ukupan broj operacija u što ulaze tijelo petlje, ali i broj inicijalizacija, provjere uvjeta i uvećavanja.

Tablica 4.1 Primjeri računanja broja operacija

Izraz	Broj operacija
`x += y;`	1
`for(i = 1; i <= n; i++)` `  x += y;`	3n + 2

`for(i = 1; i <= n; i++)`
`  for(j = 1; j <= n; j++)`
`  x += y;`
	$3n^2 + 4n + 2$

Ako pretpostavimo da imamo veliki skup podataka (`n`) na ulazu možemo vidjeti da će dominantan postati faktor uz funkciju koja najbrže raste dok će ostali izrazi imati zanemariv utjecaj. Stoga pri procjeni uzimamo u obzir jedino najbrže rastući član. U većini analiza zanemarujemo utjecaj konstante uz taj član pa ako gledamo primjere koje prikazuje tablica možemo reći da vrijeme trajanje ovisno o n raste s 1 (konstanta), n i n^2.

U matematici za ograničenje funkcije kada njen argument teži u pojedinu vrijednost ili beskonačnost koristi se obilje notacija koje se nazivaju Bachmann-Landau notacijama ili asimptotskim notacijama. U računarstvu prije svega koristimo veliko *O* notaciju za klasificiranje algoritama prema vremenu izvršavanja i utrošku memorije (prostora).

O notacija

Veliko *O* notacija karakterizira funkcije u skladu s njihovim brzinom rasta: različite funkcije s istom brzinom rasta mogu biti predstavljene istom *O* notacijom. Ovu notaciju koristimo kada želimo odrediti gornju granicu složenosti algoritma odnosno želimo odrediti koliko će biti sporo trajanje izvođenja.

Formalno definiramo *O* notaciju na slijedeći način. Kažemo da je $f(n) = O(g(n))$ ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi $f(n) \leq c|g(n)|$ za sve $n \geq n_0$. Iako mnoge funkcije mogu ispuniti ovaj uvjet cilj je pronaći najmanji *g(n)* za koji to vrijedi.

Slika 4.1 prikazuje funkciju *f(n)* i *cg(n)* koja je njezina gornja granica. Možemo vidjeti da funkcija *cg(n)* nije svugdje iznad funkcije *f(n)*, ali da za vrijednosti $n \geq n_0$ je to uvijek slučaj.

Kao primjer za izračunavanje gornje granice možemo promatrati algoritam za koga smo izračunali da je $5n^3 + 25n^2 + 77n$ koraka. Ako pogledamo definiciju, možemo uočiti da je član koji dominira n^3 (npr. za n = 100, n^3 = 1 000 000, a n^2=10 000 pri čemu vrijednosti n vrlo brzo postaju dominantnije od vrijednosti konstanti ispred potencija). Za jako velike n, iznosi ostalih članova izraza bit će zanemarivi stoga možemo reći da je složenost $O(n^3)$.

Općenito možemo pokazati slijedeće. Ako je vrijeme izvođenja određeno polinomom:

$$A(n) = a_m n^m + \cdots + a_1 n + a_0$$

složenost ovoga algoritma je $O(n^m)$.

Da to vrijedi možemo pokazati na slijedeći način.

Ako uzmemo

$$A(n) = a_m n^m + \cdots + a_1 n + a_0$$

Onda vrijedi:

$$|A(n)| = |a_m| n^m + \cdots + |a_1| n + |a_0|$$

$$|A(n)| \leq (|a_m| + |a_{m-1}|/n \ldots + |a_1|/n^{m-1} + |a_0|/n^{m-1})\, n^m$$

$$|A(n)| \leq (|a_m| + \cdots + |a_1| + |a_0|)n^m \text{ za svaki } n \geq 1$$

uz

$$c = |a_m| + \cdots + |a_1| + |a_0|$$

i

$$n_0 = 1$$

tvrdnja je dokazana.

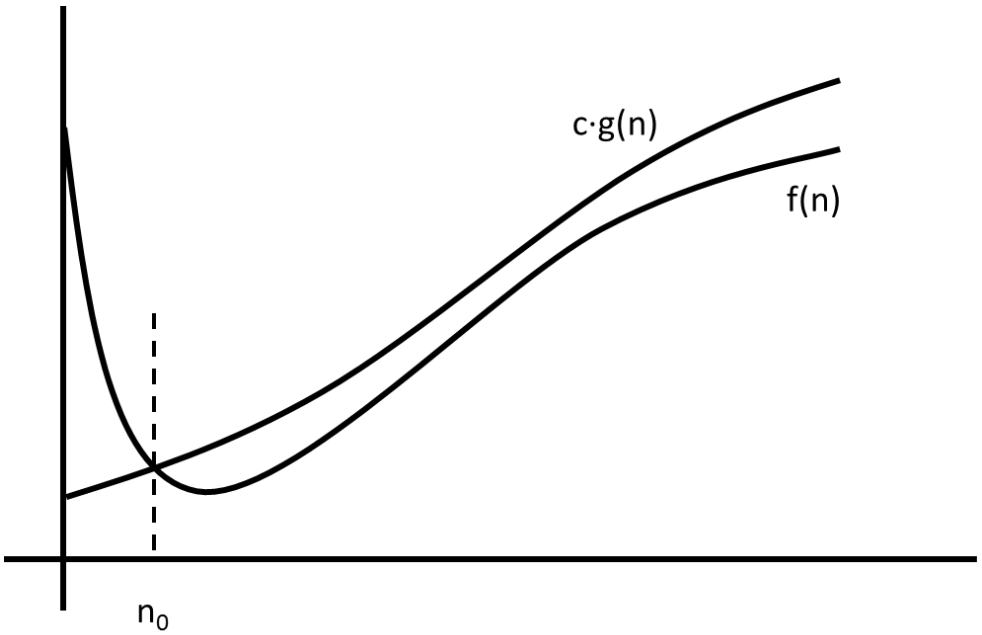
Ovim dokazom s pokazali da postoji $n_0 = 1$ i za vrijednosti n veće te će vrijediti gornja granica kao i *c* koji je apsolutna suma svim svih koeficijenata. Na ovaj način možemo za bilo koji polinom i bilo koje konstante ispred pojedinih članova dokazati da je gornja granica njegov najbrže rastući član.

Vrijednost *n* o kojoj ovisi složenost je dimenzija ulaza u naš algoritma. Najčešće je to broj članova polja ili neka druge slične struktura podataka.

Vrlo često imamo situaciju gdje imamo takozvane ugniježdene petlje kao u ovom primjeru

```
for(i = 1; i <= n; i++)
    for(j = i; j <= n; j++)
        x += y;
```

U ovom slučaju tijelo druge petlje će se izvršiti prvo jedanput pa dvaput pa tako sve do n. Ako zbrojimo broj koraka dobijemo $1 + 2 + 3 + \ldots + n$. Možemo prepoznati aritmetički red za čiju sumu znamo da iznosi $(n(n+1))/2$ što odgovara složenosti $O(n^2)$.



Slika 4.1 Prikaz odnosa funkcija *f(n)* i *cg(n)*. Za $n \geq n_0$ *f(n)* ne raste brže od *cg(n)*. Iako za manje vrijednosti od n_0 to nije slučaj, nama je važno samo da postoji dovoljno velik n_0 nakon kojeg navedeno vrijedi.

Ponekad u petljama ne radimo uvećavanje nego množenje ili dijeljenje varijable kroz koju iteriramo. U slijedećem primjeru

```
for(i = n; i >= 1; i/10)
      x += y;
```

u svakom koraku umanjujemo vrijednost `i` 10 puta. Znači prvi put će ta vrijednost biti manja 10 puta, drugi put 100, treći 1000. Matematička funkcija koja odgovara takvom brzom padu je logaritam. Stoga u ovom slučaju složenost je $O(log_{10}n)$. Važno je primijeti da svaku logaritamsku složenost možemo svesti na logaritam po bazi 2. U ovom slučaju

$$log_{10}n = \frac{log_2 n}{log_2 10} = clog_2 n = O(log_2 n)$$

Pri čemu je *c* konstantan iznos jednak recipročnoj vrijednosti logaritma od 10 po bazi 2.

Složenost u O notaciji možemo poredati po brzini rasta za dovoljno veliki *n* na slijedeći način

$$O(1) < O(log_2 n) < O(\sqrt{n}) < O(n) < O(nlog_2 n) < O(n^2) < O(n^3) < \cdots < O(2^n) < O(n!)$$

Važnije složenosti iz ovoga primjera su:

- Konstantna složenost *O(1)*. Kod ove složenost vrijeme izvršavanja ne ovisi o broju elemenata na ulazu. Ovo je najbolji mogući slučaj kojeg u praksi rijetko srećemo. Kao primjer možemo uzeti nalaženja vrijednosti člana polja ako znamo njegov indeks.
- Sublinearne složenosti kao što su $O(\sqrt{n})\ i\ O(log_2 n)$. Ova klasu složenosti imaju vrlo efikasni algoritmi kod kojih je vrijeme izvršavanja kraće od vremena koje nam treba da pročitamo cijeli ulaz.
- Linearna složenost *O(n)* je složenost koja je proporcionalna vremenu učitavanja ulaza. Ovo je također efikasna složenost, a kao primjer možemo uzeti pretraživanje na kojoj se poziciji u polju nalazi određeni element.
- Superlinearne složenosti kao što su $O(nlog_2 n), O(n^2)$ i $O(n^3)$ s naglaskom da funkcije nisu brže rastuće od polinomnih. Kod ovih složenosti je vrijeme izvršavanja dulje od linearnog, no za do vrijednosti $O(n^2)$ se smatra da su dovoljno dobre za implementaciju na računalu.
- Eksponencijalna složenost poput $O(2^n), O(n!)$ i $O(2^n)$. Ove složenosti nisu praktične za upotrebu u računala za vrijednosti *n* koji su iznad od par desetina. Algoritmi koji zahtijevaju eksponencijalno vrijeme za *n* koji je veći od 100 postaju nerješivi u razumnom vremenu, bez obzira na brzinu slijednog računala.

Tablica 4.2 prikazuje usporedbu računanja problema s algoritmima različite složenosti, te kakav će utjecaj na veličinu problema kojeg možemo riješiti ima računalo boljih performansi. Možemo vidjeti da dok za sublinearne složenosti jačina računala eksponencijalno doprinosi u veličini problema kojeg možemo riješiti u istoj jedinici vremena, kod eksponencijalne složenosti možemo riješiti neznatno veći problem od onoga što smo mogli računalom koje je i do 1000 puta slabije.

Tablica 4.2 Usporedni prikaz veličine problema kojeg možemo riješiti za algoritme koji rješavaju problem u različitoj složenosti i za računala različitih brzina

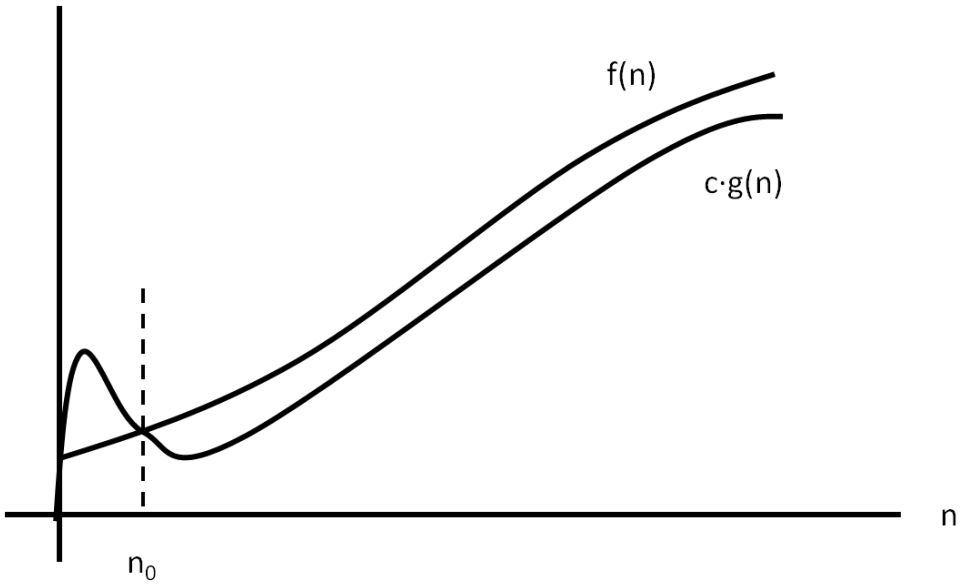
Vremenska složenost	Veličina problema koje možemo riješiti danas raspoloživim računalom	Veličina problema koje možemo riješiti 100 puta bržim računalom	Veličina problema koje možemo riješiti 1000 puta bržim računalom
$\log_2 n$	N_1	N_1^{100}	N_1^{1000}
n	N_2	100 N_2	1000 N_2
n^2	N_3	10 N_3	31.6 N_3
n^3	N_4	4.64 N_4	10 N_4
2^n	N_5	N_5 + 6.64	N_4 + 9.97
3^n	N_6	N_6 + 4.19	N_5 + 6.29

Ω notacija

Iako u analizama složenosti algoritma dominira *O* notacija, ponekad nam je važna informacija od koje familije funkcija trajanje neće biti brže. Za to koristimo Ω notaciju.

Formalno definiramo Ω notaciju na slijedeći način. Kažemo da je $f(n) = \Omega(g(n))$ ako postoje dvije pozitivne konstante c i n_0 takve da vrijedi $f(n) \geq c|g(n)|$ za sve $n \geq n_0$. Iako mnoge funkcije mogu ispuniti ovaj uvjet cilj je pronaći najveći *g(n)* za koji to vrijedi.

Ponekad kao u primjeru traženja broja u polju nesortiranih *n* brojeva gornja i donja granica neće biti jednake, odnosno $O(n) \neq \Omega(1)$. U najboljem slučaju možemo pronaći traženi broj unutar prvih par brojeva, no isto tako postoji mogućnost da ćemo morati pregledati ih sve.

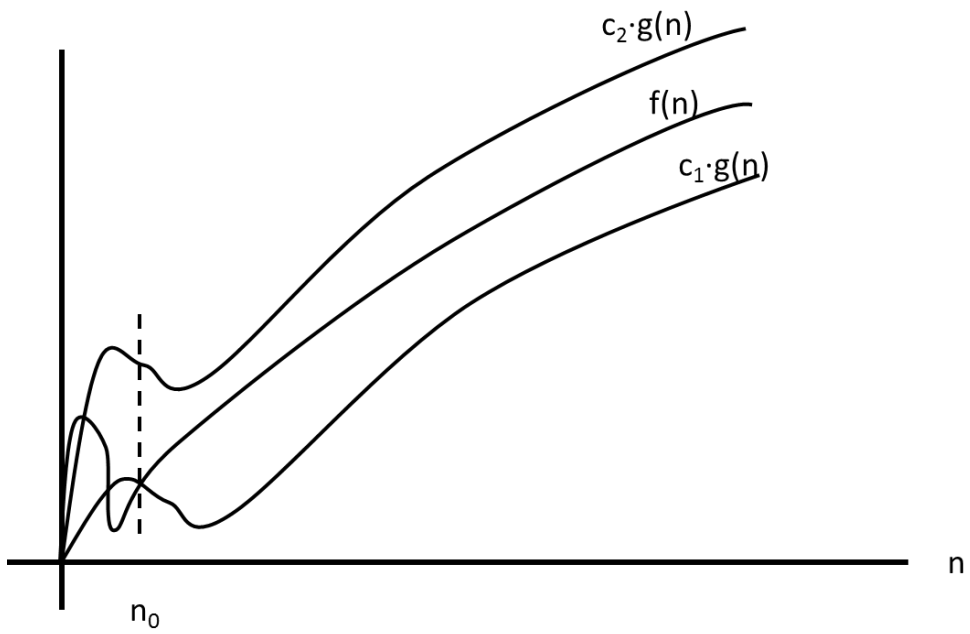


Slika 4.2 Prikaz odnosa funkcija *f(n)* i *cg(n)*. Za $n \geq n_0$ *f(n)* ne raste sporije od *cg(n)*. Iako za manje vrijednosti od n_0 to nije slučaj, nama je važno samo da postoji dovoljno velik n_0 nakon kojeg navedeno vrijedi.

Θ notacija

U nekim situacijama donja i gornja granica će biti iste. Recimo da moramo *n* puta isprintati istu poruku.

Formalno definiramo Θ notaciju na slijedeći način. Kažemo da je $f(n) = \Theta(g(n))$ ako postoje pozitivne konstante c_1, c_2 i n_0 takve da vrijedi $c_1|g(n)| \leq f(n) \leq c_2|g(n)|$ za sve $n \geq n_0$. Ovo vrijedi kada funkcije *f* i *g* rastu jednako brzo za veliki *n*.



Slika 4.3 Prikaz odnosa funkcija *f(n)*, $c_1g(n)$ i $c_2g(n)$. Za $\boldsymbol{n \geq n_0}$ ne raste sporije od $c_1g(n)$ i brže od $c_2g(n)$. Iako za manje vrijednosti od n_0 to nije slučaj, nama je važno samo da postoji dovoljno velik n_0 nakon kojeg navedeno vrijedi.

Slika 4.3 prikazuje odnos triju funkcija pri čemu je funkcija kojoj želimo odrediti složenost omeđena odozgo i odozdo s dvije funkcije koje spadaju u istu klasu, odnosno razlikuju se samo za iznos konstante ispred najbržerastućeg člana. U tome slučaju možemo odrediti takozvanu usku granicu (engl. tight bound) i koristimo Θ notaciju.

Asimptotsko vrijeme izvođenja

Kada uspoređujemo dva algoritma koji imaju istu složenost, ako želimo utvrditi koji je bolji zanimat će nas i konstanta ispred najbrže rastuće funkcije. Mjeru kojom to mjerimo nazivamo asimptotsko vrijeme izvođenja.

Ako se vratimo na jednostavan primjer:

```
for(j = i; j <= n; j++)
        x += y;
```

i korak po korak izračunamo ukupni broj operacija:

(1) Inicijalizacija i = 1 obavit će se točno jedanput.

(2) Ispitivanje uvjeta da li je i <= n, obavit će se n + 1 puta pri čemu će se prvih n puta ući u tijelo petlje dok u zadnjem koraku uvjet neće biti ispunjen i preskočiti ćemo izvršavanje.

(3) Uvećanje i++ će se obaviti točno n puta.

(4) Operacija unutar petlje će se obaviti točno n puta. Spomenuli smo da smo radi jednostavnosti pretpostavili da svaka naredba se računa kao 1 korak. U ovome slučaju se radi zapravo o dvije operacije zbrajanja i pridruživanja, no radi jednostavnosti držimo se toga da jednu naredbenu liniju računamo kao jednu operaciju.

Ako zbrojimo (1), (2) i (3) dobivamo 3n + 2 operacija. S obzirom da je najbrže rastući član n možemo reći da je asimptotsko vrijeme izvođenja u ovom slučaju 3n.

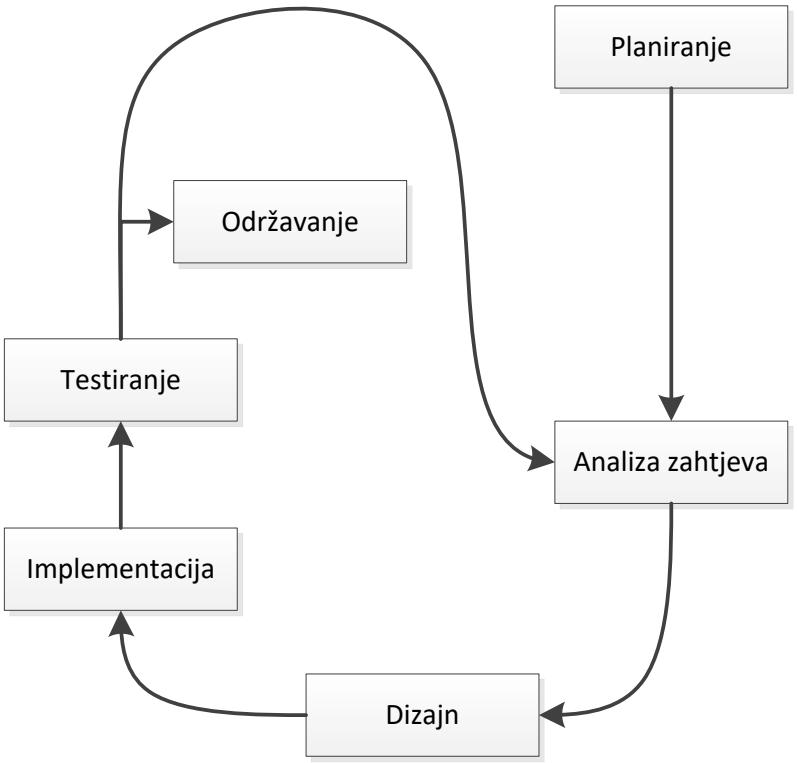
5 Procesi programskog inženjerstva

5.1 Proces oblikovanja programske potpore

Programsko inženjerstvo (engl. *software engineering*) je disciplina koja se bavi svim aspektima razvoja i održavanja programskog proizvoda (engl. *software product*). Organizacija IEEE programsko inženjerstvo definira kao metodologiju primjene sustavnih, discipliniranih i mjerljivih pristupa razvoju programske potpore (engl. *software*). Programsko inženjerstvo važno je jer proučava inženjerska načela razvoja složenih sustava u domeni razvoja programske potpore i daje specifične smjernice na sličan način kao što to u svojim područjima imaju i druge inženjerske discipline (poput npr. građevine, elektrotehnike, strojarstva itd.).

Potreba za razvojem programskog inženjerstva kao discipline i metodologije došla je sa složenošću programske potpore i teškoćama upravljanja razvojem i održavanjem programskih proizvoda, koji su s razvojem tehnologije postajali sve kompleksniji. Zbog neodgovarajuće metodologije upravljanja procesom oblikovanja i razvoja programske potpore veliki broj razvojnih projekata ostaje nedovršen, ili ne ostvari zadane ciljeve na zadovoljavajući način, odnosno zbog lošeg upravljanja procesom kasnije može uzrokovati enormne financijske gubitke i druge vrste šteta. Poznati su slučajevi poput npr. projekta *FBI Virtual Case File System*, koji je trebao osigurati modernizaciju procesa i načina rada te službe, ali koji je zbog lošeg upravljanja procesom razvoja programskog rješenja nakon 5 godina razvoja, 700,000 linija programskog koda i $170 milijuna dolara uloženih u razvoj 2005. godine napušten. Drugi poznati primjer je neuspjelo lansiranje Ariane 5 rakete u okviru *Ariane* projekta Europske svemirske agencije, gdje je trivijalna programska pogreška uzrokovala eksploziju letjelice 4. lipnja 1996. svega 40 sekundi nakon lansiranja, a u cijeli projekt bilo je tada uloženo oko $7 milijaradi dolara, od čega samo u letjelicu oko $500 milijuna dolara. Pogreška je nastala uslijed pogrešne pretvorbe 64-bitnog broja u aritmetici pomičnog zareza u 16-bitnu cjelobrojnu vrijednost, što je primjer do kakvih posljedica može dovesti neodgovarajuće upravljanje ciklusom razvoja programske potpore, u kojeg nije bila uključena odgovarajuća metodologija testiranja konačnog produkta. Programsko inženjerstvo kao disciplina ima za cilj izbjeći takve primjere i upravljati procesom razvoja programske

potpore na sustavan i discipliniran način kako bi se proizveo konačni rezultat usklađen s potrebama korisnika i s pouzdanom funkcionalnošću, s održivim planom dugoročne podrške i nadogradnje rješenja.



Slika 5.1 Ciklus životnog vijeka razvoja programske potpore

Slika 5.1 prikazuje glavne aktivnosti u okviru procesa razvoja programske potpore. Aktivnosti obuhvaćaju planiranje, analizu zahtjeva, projektiranje, implementaciju, testiranje i održavanje.

Planiranje predstavlja prvu, najraniju faza u razvoju programskog rješenja u kojoj se na temelju inicijalne specifikacije donosi okvirna procjena opsega i troškova projekta. Rezultat ove faze tipično je plan upravljanja projektom (engl. *Software Project Management Plan*, SPMP), u kojem se identificiraju krovne aktivnosti, radni paketi, vremenski raspored i resursi, a na temelju kojih se donosi okvira procjena troškova razvoja prema naručitelju. SPMP plan napravljen u ovoj početnoj fazi važan je prilikom poslovnog ugovaranja, a uobičajeno je da ga se nadograđuje u kasnijim fazama kako se prikuplja više specifičnih informacija od strane korisnika.

U fazi **analize zahtjeva** prikupljaju se detaljne informacije o potrebama korisnika i problemu za kojeg se razvija programsko rješenje. Dok se u fazi planiranja prikupljaju samo minimalne informacije potrebne za početak planiranja projekta, u fazi analize zahtjeva prikupljaju se sveobuhvatne informacije vezane uz funkcionalnost i mogućnosti programa, uz dodatne informacije o ciljanim performansama, pouzdanosti i načinu korištenja. Zahtjevi opisuju **što** programska potpora treba raditi, a to je preduvjet za sljedeće faze koje odgovaraju na pitanje **kako** razviti programsku potporu koja će to zaista i raditi. Rezultati analize zahtjeva opisani su formalno kroz dokument specifikacije zahtjeva na programsku potporu (engl. *Software Requirements Specification*, SRS).

Nakon prikupljanja zahtjeva slijedi faza **dizajna** programske potpore u kojoj se definira na koji način će se graditi programska potpora koja mora zadovoljiti zahtjeve definirane u drugom koraku. U ovoj fazi naglasak je na *arhitekturi* programske potpore, koja definira od kojih se podsustava i komponenti sastoji cjelovito rješenje, koje će se tehnologije koristiti, kako će komponente biti međusobno povezane, koja su sučelja među komponentama i prema vanjskim sustavima i sl. Izlaz iz ove faze je dokument dizajna programske potpore (engl. *Software Design Document*, SDD), koji predstavlja ulaz za fazu implementacije, a koji mora biti usklađen sa specifikacijom zahtjeva na programsku potporu (SRS).

U fazi **implementacije** razvija se programska potpora na temelju SDD dokumenta u odabranom programskom jeziku (ili više njih) i u odabranim tehnologijama. Ovo je faza u kojoj se svi ranije navedeni zahtjevi kodiraju u programski jezik razumljiv računalu, a izlaz iz ove faze je programski kôd spreman za testiranje.

U fazi **testiranja** programski kôd se testira kako bi se ustanovila njegova ispravnost i usklađenost s dokumentom zahtjeva na programsku potporu (SRS). Testiranje se sastoji od tri razine:

- testiranje tijekom razvoja, koje provode programeri koji razvijaju programsku potporu; u ovoj fazi testiranja ekstenzivno se koristi testiranje osnovnih jedinica programske potpore (engl. *Unit Testing*) kojem je zadaća osigurati ispravnost gradivnih elemenata rješenja

- testiranje na razini povezivanja modula, kako bi se testirala sučelja u povezivanju među programskim modulima i ustanovila ispravnost rada u njihovom međudjelovanju
- testiranje na razini cjelokupnog rješenja nakon integracije svih modula u jednu cjelinu (integracijsko testiranje)

Kada je moguće, preporučljivo je provesti fazu *beta testiranja*, u kojoj se krajnjem korisniku daje programska potpora koja je vrlo blizu završetka kako bi se provela dodatna testiranja u okolnostima koje što bliže stvarnim uvjetima u kojima će se koristiti programska potpora. Nakon *beta testiranja* slijedi finalno testiranje prihvatljivosti (engl. *acceptance testing*), koje nakon što uspješno prođe rezultira puštanjem dovršene verzije programske potpore korisniku (engl. *final release*).

Održavanje programske potpore započinje nakon puštanja konačne verzije i traje tijekom cijelog životnog ciklusa programske potpore. U ovoj fazi česte su nadogradnje programske potpore uslijed (1) pogrešaka uočenih od strane korisnika tijekom normalnog korištenja programske potpore, (2) naknadnih zahtjeva naručitelja programske potpore i (3) ažuriranja funkcionalnosti od strane razvojnog tima s ciljem unaprijeđenja funkcionalnosti ili performansi programske potpore.

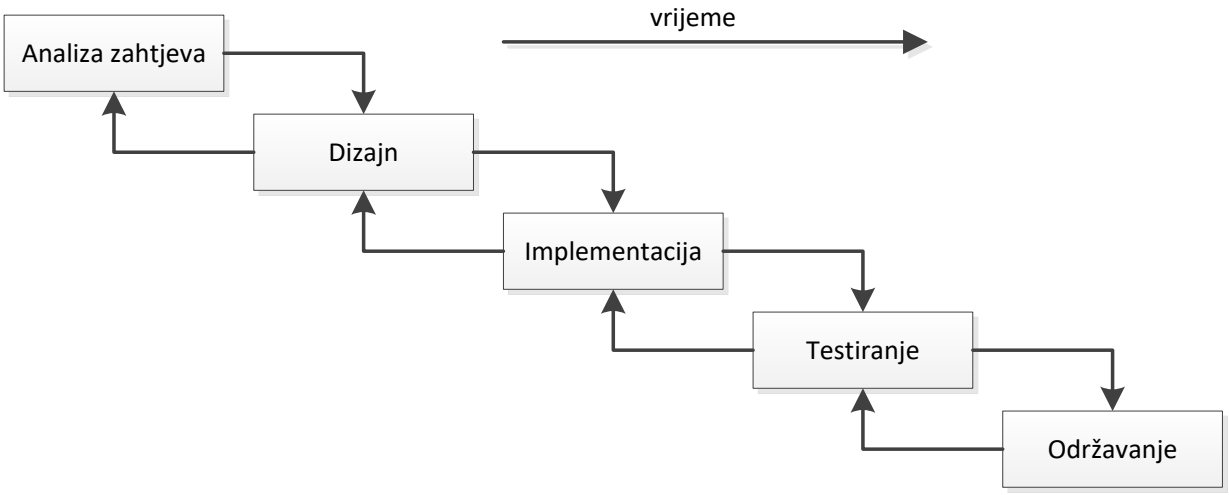
5.2 Modeli procesa programskog inženjerstva

Procesi programskog inženjerstva definiraju redoslijed i učestalnost pojedinih generičkih faza u oblikovanju programske potpore. Ti procesi usko su povezani s upravljanjem projekta razvoja programske potpore, a mogu se podijeliti na sljedeće modele:

- vodopadni (engl. *waterfall*)
- iterativni i inkrementalni razvoj (engl. *iterative and incremental development*)
- spiralni model (engl. *spiral*)
- UP (engl. *unified process*)
- agilni (engl. *agile*)

Vodopadni (engl. *waterfall*) model najstariji je model procesa programskog inženjerstva, koji pretpostavlja sekvencijalno odvijanje aktivnosti. Svoje korijene vuče iz tradicionalnih

industrija (proizvođačke i građevinske industrije), u kojima i danas predstavlja dominantni pristup upravljanja procesom, ali se pokazalo da zbog specifičnosti razvoja programske potpore preslikavanje takvog tradicionalnog modela pogodnog za materijalne proizvodno-intenzivne aktivnosti često nije dobar odabir za razvoj programske potpore koja se razvija na suštinski drugačiji način.

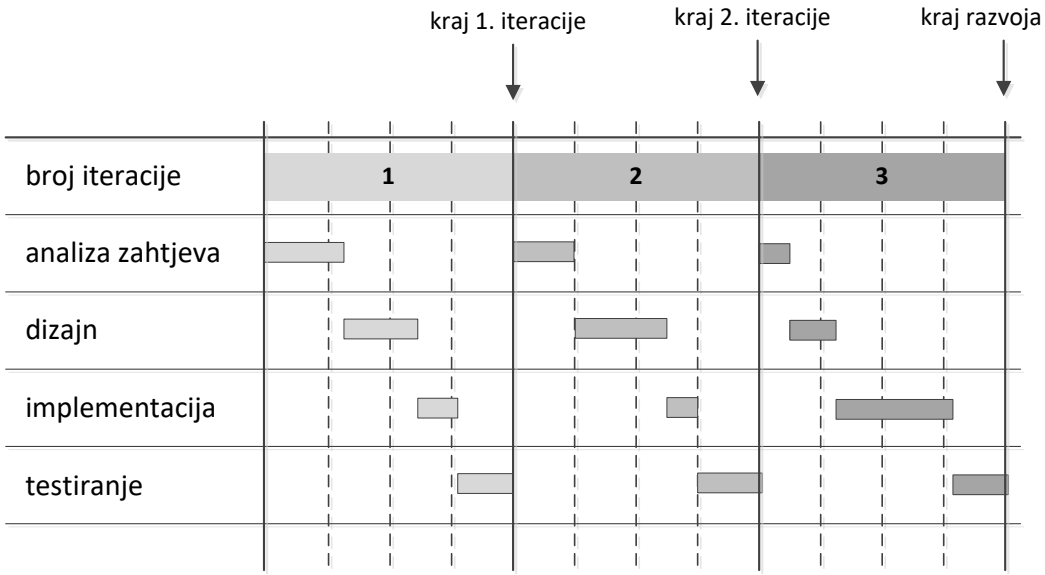


Slika 5.2 Vodopadni model procesa (engl. *waterfall*)

Na slici 5.2 prikazan je vodopadni model procesa u kojem se faze analize zahtjeva, dizajna, implementacija, testiranja i održavanja izvode slijedno jedna za drugom. Prednosti vodopadnog modela su jednostavnost korištenja metodologije, veliko iskustvo u ovakvom pristupu u domeni tradicionalnih radno-intenzivnih industrija, jednostavnost upravljanja samim procesom zbog rigidnosti modela i dobro je primjenjiv na manje projekte kod kojih nije potrebno uvoditi više iteracija i kod kojih su zahtjevi dobro unaprijed definirani. Nedostataci ovog modela su da je potrebno unaprijed dobro definirati sve zahtjeve (što je često nemoguć zadatak kod složenijih projekata), nedostatak povratnih veza i iteracija u procesu razvoja, nemogućnost ranog otkrivanja sustavnih pogrešaka, nedostatak paralelizma i neefikasna uporaba resursa.

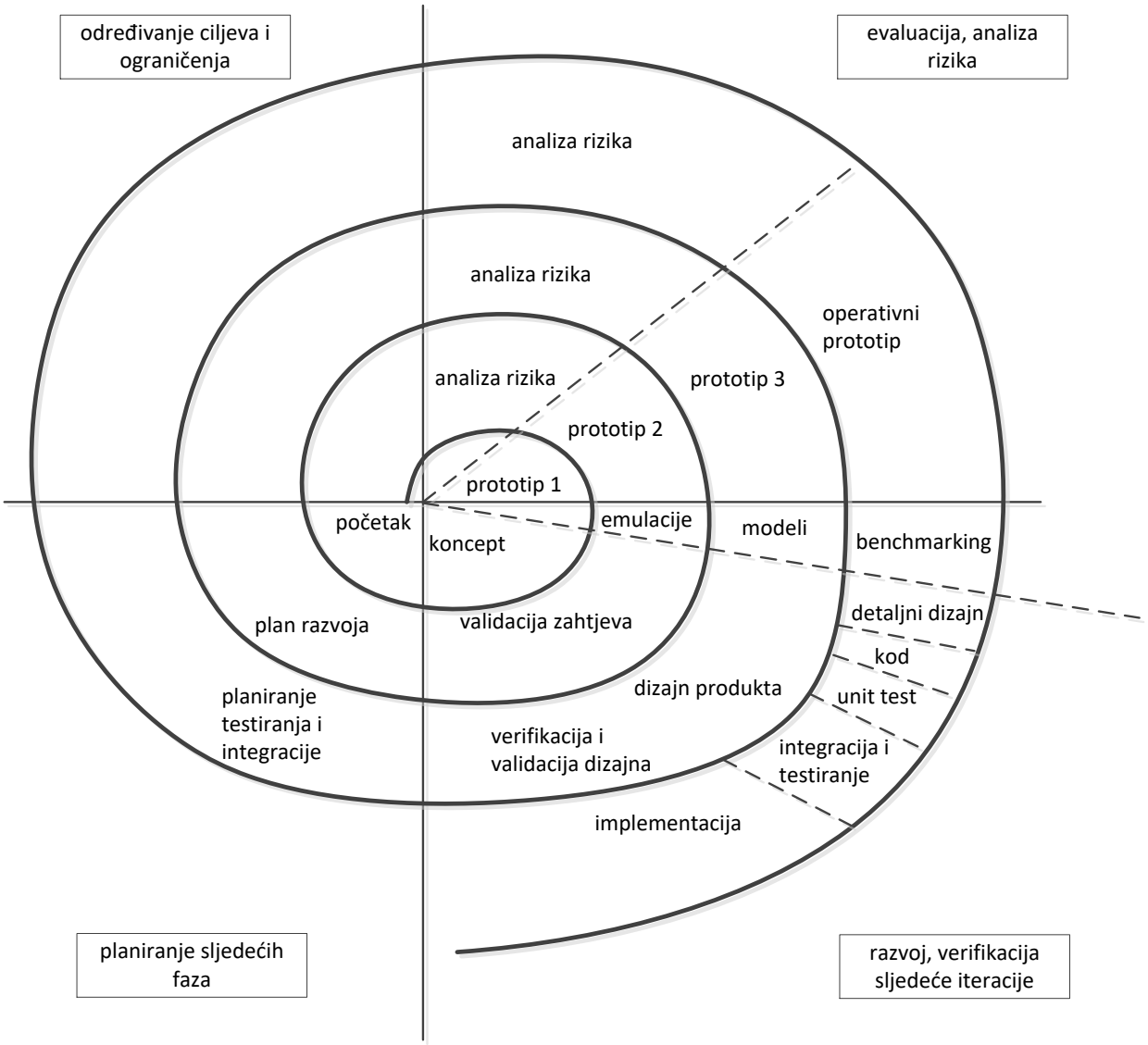
Iterativni (engl. *interative*) model predstavlja proces koji se temelji na višestrukom ponavljanju cjelokupnog ciklusa procesa razvoja programske potpore. Kada se tijekom svake iteracije ostvare razmjerno maleni pomaci, govori se o **inkrementalnom** (engl. *incremental*)

razvoju programske potpore. Primjer nadogradnje vodopadnog modela iterativnim procesom kroz niz cikličkih iteracija jednog cjelokupnog ciklusa vodopadnog modela prikazan je na slici 5.3.



Slika 5.3 Iterativni model procesa (engl. *iterative*) za primjer od tri iteracije

Varijanta nadogradnje iterativnog je tzv. **spiralni model** (Barry Boehm), čiji je ciklus prikazan na slici 5.4. Glavna značajka ovog modela je da se radi o procesu u kojeg je duboko ugrađeno upravljanje rizicima kako bi se umanjila mogućnost da projekt neuspješno završi. Razvoj programke potpore odvija se po cikličkim iterativnim fazama koje se približavaju konačnom rješenju kako projekt napreduje. Prednost ovog modela je da se rizicima upravlja unutar samog procesa, da se programska potpora razvija kako projekt napreduje i da je planiranje uključeno kao dio iterativnog procesa. Nedostaci ovakvog spiralnog modela su složenost u praktičnoj primjeni i potreba za uvođenjem nepotrebnih koraka i poslova kada se radi o manjim i jednostavnijim projektima.

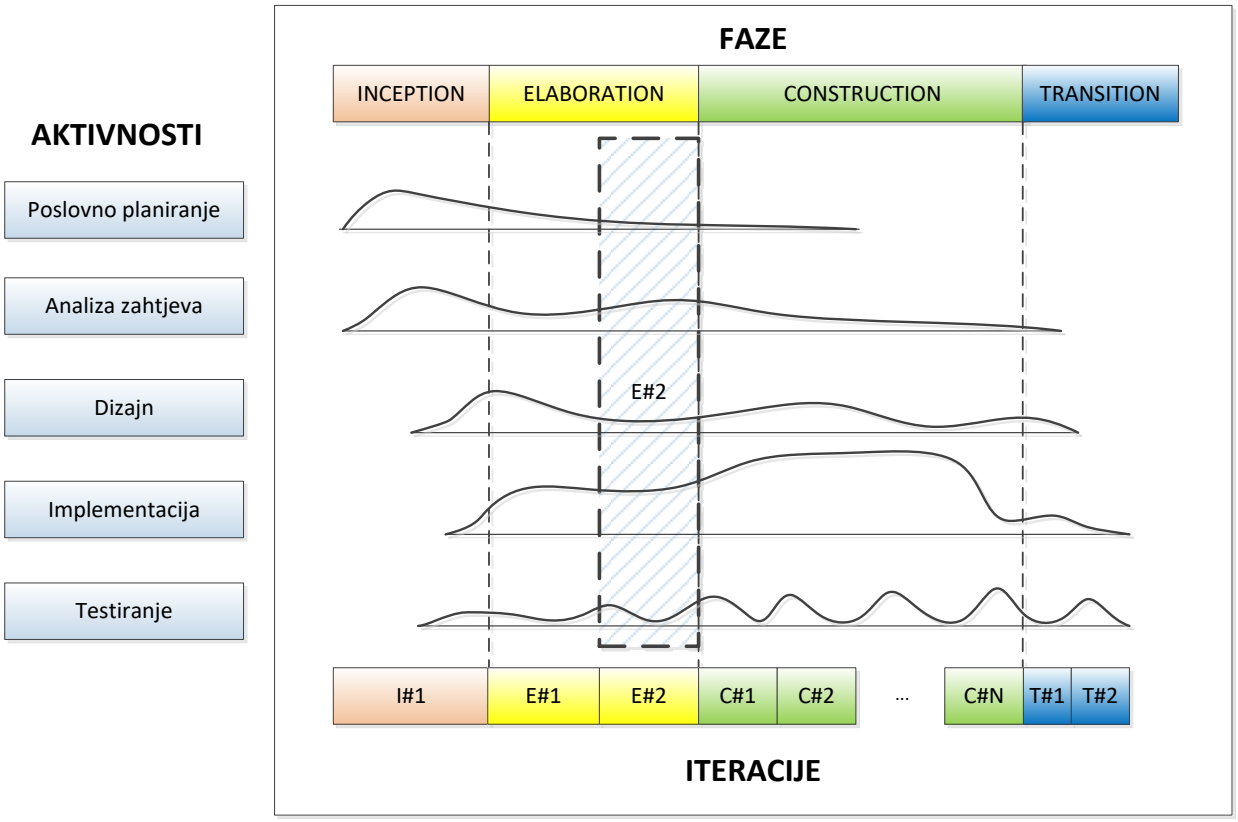


Slika 5.4 Boehmov spiralni model procesa

Unified Process (UP) je varijanta iterativnog i inkrementalnog procesa razvoja programske potpore koja se temelji na razmatranju korisničkih slučajeva (engl. *use-cases*), odnosno obrazaca uporabe produkta te fokusiranosti na arhitekturu (engl. *achitecture-centric*), uz korištenje formalnih opisa u specifikaciji temeljenih na primjeni UML jezika za modeliranje (engl. *Unified Modeling Language*). Proces se sastoji se sastoje od četiri faze, koje su podijeljene na iteracije:

- početak (engl. *inception*) – studija izvedivosti, razrada poslovnog modela, vizija produkta, procjena opsega razvoja, procjena troškova, izrada okvirnog vremenskog rasporeda aktivnosti, analiza rizika, izrada prototipa
- razrada (engl. *elaboration*) – detaljnija razrada specifikacije programske potpore, dizajn arhitekture rješenja, implementacija jezgre arhitekture rješenja, dodatna procjena rizika, priprema detaljnog plana za implementaciju rješenja
- razvoj (engl. *construction*) – programiranje, dovršetak implementacije započetog prototipa, testiranje i priprema za puštanje produkta
- tranzicija (engl. *transition*) – provođenje beta testiranja, ispravljanje uočenih pogrešaka, izrada korisničkih uputa, priprema konačnog produkta, edukacija korisnika.

Na slici 5.5 prikazan je primjer razvoja temeljenog na UP procesu. Svaka faza sastoji se od jedne ili više iteracija, a u svakoj iteraciji provode se aktivnosti koje su inače sukcesivne i bez povratnih veza u klasičnom vodopadnom modelu. Na slici je grafom prikazano procjenjeno opterećenje pojedinih tipova aktivnosti za svaku iteraciju, pri čemu je vidljivo da u nekim fazama i iteracijama nema nekih aktivnosti (npr. u *transition* fazi nema više poslovnog planiranja). Na slici je istaknut primjer druge iteracija faze razrade (*elaboration*) gdje se vidi da su sve aktivnosti prisutne, ali s različitim intenzitetom.



Slika 5.5 *Unified Process* (UP) iterativni proces programskog inženjerstva

Prednosti UP procesa su da je puno fleksibilniji od klasičnog vodopadnog modela na domenu razvoja programske potpore jer je bolje usklađen sa stvarnim okruženjem i prilikama u kojima se razvijaju programski produkti, UP proces je inkluzivan u smislu da je pogodan za gotovo sve vrste projekata razvoja programske potpore i da se radi o zrelom modelu koji se pokazao uspješnim u praksi. Osnovni nedostatak UP modela je da je izvorno zamišljen za složene projekte i da postoje drugi moderni pristupi koji su bolji odabir za manje i jednostavnije projekte.

Agilni procesi nastali su kao odgovor na u praksi potvrđenu neefikasnost i složenost primjene tradicionalnih modela programskog inženjerstva, pogotovo na projekte niže i srednje razine kompleksnosti. Značajniji razvoj i popularizacija agilne metodologije započeli su 2001. kada se oformila grupa *Agile Alliance*, koja je nove smjernice kako razvijati programsku potporu dala u dokumentu *Manifesto for Agile Software Development*. Agilne

metode programskog inženjerstva su one koje svoje postavke temelje na najbitnijim odrednicama tog dokumenta, kao što su primjerice:

- usmjerenost na iterativni i inkrementalni razvoj,
- rad u malim, dobro povezanim timovima,
- redovito i često sastajanje s korisnicima tijekom procesa razvoja,
- usmjerenost na kôd, dokumentacija minimalna,
- kontinuirano testiranje osnovnih jedinica (engl. *unit-testing*),
- preferiranje osobnih interakcija nad procesima i alatima, funkcionalnog softvera nad opširnom dokumentacijom, suradnje s korisnikom nad ugovornim pregovaranjem, upravljanjem promjenama nad praćenjem zacrtanog plana itd.

Prednosti agilnih procesa su da je povećana motivacija razvojnog tima, postojanje neke verzije funkcionalne programske potpore tijekom cijelog procesa razvoja i bolji doprinos korisnika procesu zbog čvršće uključenosti u sam razvojni proces. Nedostaci su da je takav pristup često neodgovarajući za velike i složene aplikacije te da je upitna kvaliteta i opseg dokumentacije koja slijedi programski produkt jer je dokumentiranje stavljeno u drugi plan. S obzirom da su agilni procesi danas postali popularan odabir za upravljanje projektima, pogotovo za timove i projekte male i srednje veličine i složenosti, a i da se takav pristup pokazao vrlo učinkovitim u praksi, u sljedećem poglavlju bit će detaljnije opisane značajke i smjernice nekih najvažnijih agilnih procesa razvoja programske potpore.

5.3 Agilni procesi

Devedestih godina prošlog stoljeća agilni procesi pojavljaju se kao alternativa klasičnim pristupima programskog inženjerstva koji su svoje korijene vukli iz tradicionalnog načina upravljanja projektima prvenstveno povezanih s radno-intenzivnim industrijama. Osnovni problem kod tih procesa bio je taj da preslikavanje načela koja su se pokazala uspješnima u većem broju inženjerskih i tehnoloških domena nisu davala dobre rezultate u području razvoja programske potpore, prvenstveno iz razloga što za razliku od tih drugih područja na početku projekta često nije do kraja jasno što je sve potrebno razviti u dovoljnoj razini detalja, niti su unaprijed poznate sve moguće interakcije između podsustava i komponenti finalnog produkta, zbog izuzetno visoke složenosti konačnog produkta koja se pojavila s

napretkom programskih tehnologija i očekivanja korisnika aplikacija. Glavni cilj agilnih procesa bio je upravo pružiti odgovor kako upravljati razvojem projekta s nedovoljno poznatom specifikacijom koja se neprekidno nadograđuje kako se projekt razvija i koja se često dinamički mijenja u interakciji s korisnikom, a koji postaje i jedan od važnih sudionika samog procesa razvoja programske potpore.

Agilni razvoj programske potpore ne odnosi se isključivo na jednu metodu, već je skupni naziv za različite procese koje slijede najvažnije odrednice *Agile Manifesta*. Tri popularna primjera agilnih procesa bit će prikazana u nastavku:

- *Extreme Programming* (XP)
- *Lean Software Development*
- *Scrum*

***Extreme Programming* (XP)** metodologiju izvorno je razvio 1996. Kent Beck u DaimlerChrysleru s ciljem da olakša i učini efikasnijim proces programiranja, kao odgovor na nezadovoljstvo tradicionalnim pristupima, pogotovo vodopadnim procesom. Četiri su temeljne „vrijednosti“ na kojima se temelji XP pristup:

- *komunikacija* – prisutsvo korisnika u procesu, programiranje u paru, pridržavanje dogovornih standarda za produkciju programskog koda itd.,
- *jednostavnost* – najjednostavniji dizajn programa za trenutne specifikacije, *refactoring* koda (promjena koda radi usavršavanja ili pojednostavljenja, bez promjene vanjske funkcionalnosti),
- *povratna informacija* – kontinuirano testiranje i integracija, minimalne inkrementalne *release* verzije (s minimalnim dodanim novim mogućnostima),
- *„hrabrost“* – planiranje i procjena temeljem korisničkih priča (engl. *user stories*), održivi ritam razvoja.

XP iteracije traju 1-3 tjedna, a nakon svake iteracije producira se *mini-release* produkta koji je potpuno testiran. XP pristup preferira *kontinuiranu integraciju* pred *agregacijom razdvojenih velikih modula* na kraju procesa razvoja, kao što tipično impliciraju tradicionalni pristupi poput vodopadnog procesa. Proces planiranja u velikoj mjeri oslanja se na pristup „korisničkih priča“ (engl. *user stories*), koji predstavljaju input od strane korisnika koji su

kontinuirano uključeni u razvojne iteracije. Važan aspekt čini i tzv. *Test-Driver Development*, u kojem se potiče izrada tzv. *unit-testova* koji se pišu *prije* kôda za čije testiranje služe.

Lean Software Development proces razvili su Mary and Tom Poppendieck, a naslanja se na dobru praksu koju koriste kompanije poput npr. Toyote. *Lean* pristup fokusira se na *vrijednost* za korisnika i način kako na najefikasniji način korisniku isporučiti tu vrijednost kroz oslobađanje od svega nepotrebnog što opterećuje proces razvoja usmjeren na isporuku vrijednosti korisniku. Pristup je primjenjiv i na druge domene osim programskog inženjerstva, a osnovna načela te metodologije su sljedeća:

- eliminacija svega nepotrebnog,
- pojačano učenje,
- odloženo odlučivanje,
- što brža isporuka rješenja,
- motivacija tima,
- cjelovitost rješenja.

Eliminacija nepotrebnog foksuira se na uklanjanje iz procesa svega što ne donosi vrijednost za korisnika, a što je kod procesa programskog inženjerstva prepoznato kao npr. rad na nepotrebnim mogućnostima produkta, nepotrebno kompleksna rješenja, neefikasna komunikacija, višestruki posao na istom problemu itd. *Pojačano učenje* podrazumijeva kontinuiranu komunikaciju s korisnikom i brze cikluse testiranja i nadogradnje kako bi se moglo što prije doći do zaključka je li trenutni pravac razvoja dobar ili je potrebno nešto mijenjati, što se može bolje zaključiti kroz iterativni razvoj u kojem se inkrementalno unaprijeđuje prototip, nego kroz *a priori* pretpostavke prikupljene samo kroz dokumentirane zahjeve, a koji nisu revidirani kroz povratnu vezu praktičnih međudjelovanja različitih zahtjeva. S obzirom da je proces razvoja programske potpore prožet nesigurnošću i nedostatnim informacijama, *odloženo odlučivanje* pruža priliku da se odluke donose onda kada se iskustveno i od strane korisnika prikupi najkvalitetnija i najpotpunija informacija u svakoj iteraciji razvoja, a koja će biti to kvalitetnija što je prototip sustava iterativno i inkrementalno dalje razvijen. *Brza isporuka rješenja* važna je kako bi se ubrzale pojedine

iteracije (poput učenja i odlučivanja na temelju rezultata pojedinih kratkih iteracija), a ne samo dovršetak cjelokupnog projekta. U *motivaciji tima* važnu ulogu ima uključenje članova tima u procese donošenja odluka, umjesto naglašenog mikroupravljanja od strane voditelja. Korisnik (kupac) treba biti u konačnici zadovoljan sa sustavom u *cjelini*, što uključuje cjelokupno iskustvo programske potpore ne samo s gledišta korištenja, već i načina distribucije, cijene, marketinga, mogućnosti održavanja i nadogradnji te međudjelovanja pojedinih podsustava u rješenju.

Scrum je iterativni i inkrementalni agilni proces koji je razvijen je u prvoj polovici devedestih godina prošlog stoljeća, a u osnovi metodologije ovog procesa leže dva važna empirijska zaključka vezana uz proces razvoja programske potpore: (1) kupci (korisnici) mijenit će mišljenje o tome što zaista žele kako proces razvoja napreduje i (2) treba očekivati da će se u proces razvoja uplesti nepredvidljivi faktori koje je nemoguće predvidjeti na početku, što često uzrokuje neuspjeh softverskih projekata vođenih tradicionalnim procesima. U Scrum terminologiji razlikujemo uloge (engl. *Roles*), faze procesa ili događaje (engl. *workflow*) i artefakte (engl. *artifacts*). Uloge (*roles*) su sljedeće:

- vlasnik projekta (*product owner*) - osoba koja zastupa interese korisnika ili naručitelja razvoja programske potpore, a odgovorna je viziju produkta prenijeti razvojnom timu,
- razvojni tim (*development team*) - tim koji je odgovoran za razvoj programske potpore,
- Scrum master (*product owner*) - osoba zadužena za pomaganje timu u uklanjanju prepreka i dolasku do cilja svake iteracije; Scrum master ne vodi tim kao klasični voditelj tima (project manager), već osigurava da se tim pridržava Scrum okvira u implementaciji projekta.

Faze procesa, odnosno događaji, obuhvaćaju sljedeće pojmove:

- *Sprint* - temeljna iteracija razvoja u Scrum okviru koja je vremenski ograničena tipično na period od tjedan do mjesec dana; cilj jedne Sprint iteracije je u vremenski fokusiranom timskom naporu proizvesti upotrebljivi i zaokruženi inkrement proizvoda; svaki Sprint sastoji se od planiranja Sprinta, dnevnih Scrum sastanaka, rada na razvoju proizvoda te revizije i retrospektive Sprinta; metodologija naglašava

da bi svaki Sprint trebao rezultirati potencijalno upotrebljivim i zaokruženim inkrementom produkta, a sljedeća Sprint iteracija slijedi odmah po završetku prethodne,

- planiranje Sprinta - s obzirom da svaki Sprint predstavlja mini-projekt, na početku je važno isplanirati Sprint, u okviru čega se dogovara opseg posla i rezultati koji se žele postići u predstojećem Sprintu, a također se obavlja odabir zadataka iz *Backlog*-a koji će biti uključeni u tekući Sprint,
- dnevni Scrum sastanak (*Daily Scrum*) - svakodnevni sastanci koji imaju striktno ograničeno trajanje od 15 minuta koji služi da razvojni timovi usklade svoje aktivnosti za sljedeća 24 sata; na sastanku sudjeluju svi članovi Scrum tima, što uz Scrum mastera i članove razvojnog tima uključuje i vlasnika produkta; dnevni Scrum ne služi za rješavanje problema, već za učinkovitu komunikaciju među članovima tima, a svi problemi koje se izlože na dnevnom Scrumu i koje je potrebno posebno rješavati se rješavaju tipično unutar zainteresirane podgrupe nakon sastanka; na sastanku se iznose isključivo informacija o tome što su članovi tima napravili od zadnjeg dnevnog Scruma, što će raditi danas i jesu na kakve su poteškoće naišli od prethodnog dnevnog Scruma,
- Sprint revizija i retrospektiva (*Sprint review and retrospective*) - na kraju svakog Sprinta održavaju se dva sastanka:
    - Sprint revizija (*review*) - izvješće o poslu koji je dovršen unutar Sprinta i o poslu koji je planiran, ali nije dovršen; prezentira se trenutna završena iteracija svim dionicima (demo); tim i dionici dogovaraju se što će se raditi u sljedećem Sprintu,
    - Sprint retrospektiva (*restrospective*) - analizira se što je bilo dobro, a što loše u prethodnom Sprintu te što se može poboljšati za sljedeći.

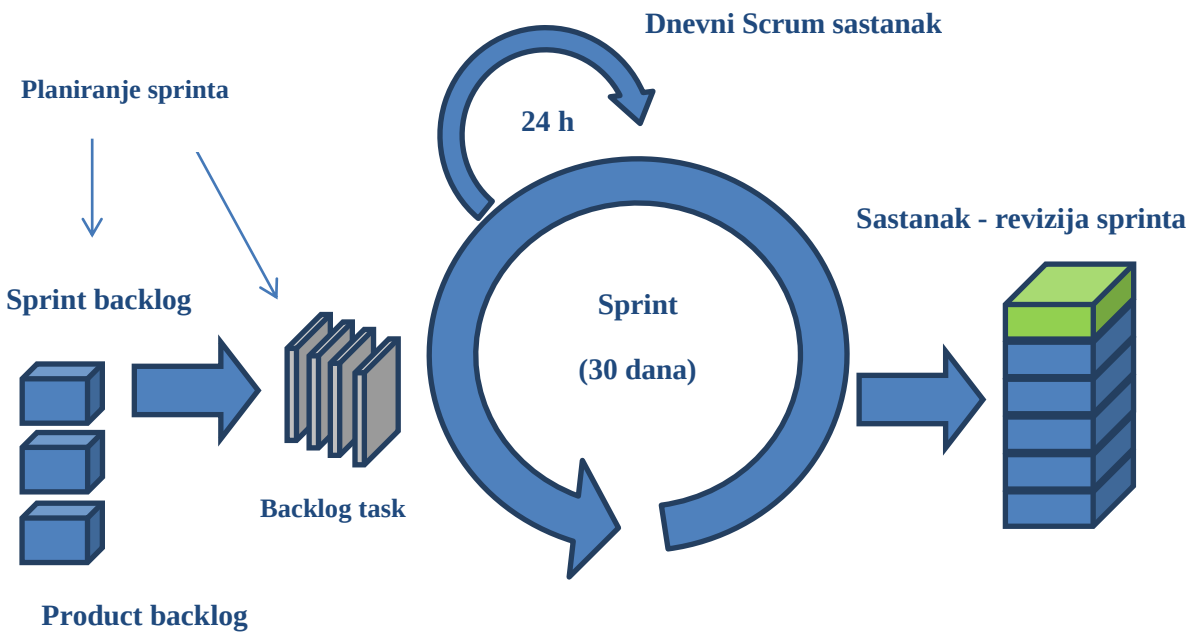
Artefakti (*articafts*) u Scrum procesu obuhvaćaju sljedeće:

- *Product backlog* - predstavlja popis svih zahtjeva koji čine specifikaciju što je potrebno napraviti da bi se došlo do konačnog produkta; sastoji se od mogućnosti programa (*features*), zahtjeva za ispravkom uočenih pogrešaka (*bug-fixes*),

nefunkcionalnih zahtjeva (npr. korisničko sučelje) itd.; vlasnik proizvoda (*product owner*) odgovoran je za izradu i održavanje Product backloga; važno je naglasiti da se sadržaj i prioriteti u Product backlogu kontinuirano evoluiraju, mijenjanju i nadopunjuju; unosi u Product backlog sortirani su po važnosti, riziku, vrijednosti i sl.

- *Sprint backlog* - predstavlja popis stavki odabranih iz Product backloga koje je potrebno realizirati u sljedećem Sprintu; ako su stavke u Product backlogu prevelike da bi ih se u cjelini uključilo u Sprint backlog, mogu se raspodijeliti na manje cjeline (radne zadatke) koji se selektivno uvrštavaju u Sprint backlog; zadaci iz Sprint backloga se *ne dodjeluju* članovima razvojnog tima, već se potiče dogovor i inicijativa među članovima tima tko će preuzeti koji zadatak, sukladno interesima, dosadašnjem tijeku razvoja i sposobnostima člana razvojnog tima; na taj način potiče se uključenost članova tima u procese odlučivanja i daje im se autonomija koja rezultira većim zadovoljstvom i motiviranošću članova tima; Sprint backlog definira ujedno što će biti inkrementalni rezultat sljedeće Sprint iteracije; važno je naglasiti da kao i kod Product backloga je moguće mijenjati sadržaj Sprint backloga tijekom trajanja Sprinta,
- inkrement proizvoda (*Product increment*) - inkrement proizvoda (koji se ponekad naziva i *Potentially Shippable Increment*, PSI) predstavlja nadogradnju proizvoda na kraju trenutne Sprint iteracije, koja u sebi uključuje sve stavke iz trenutnog Sprint backloga i stavke iz Product backloga koje su implementirane u prethodnim Sprintovima; važno je naglasiti da se uspješnom Sprint iteracijom smatra samo slučaj ako je proizvod potpuno dovršen u smislu odabranih stavki iz backloga, tj. načelno spreman za puštanje (*release*) ako vlasnik produkta tako odluči.

Na slici 5.6 prikazan je shematski raspored faza i artefakata Scrum procesa.



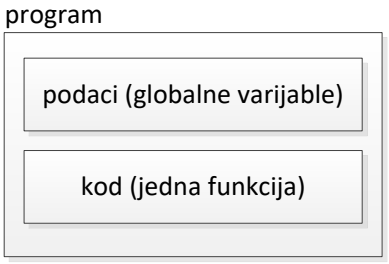
Slika 5.6 Scrum iterativni agilni proces programskog inženjerstva

6 Objektno orijentirano programiranje

6.1 Objektno orijentirana paradigma

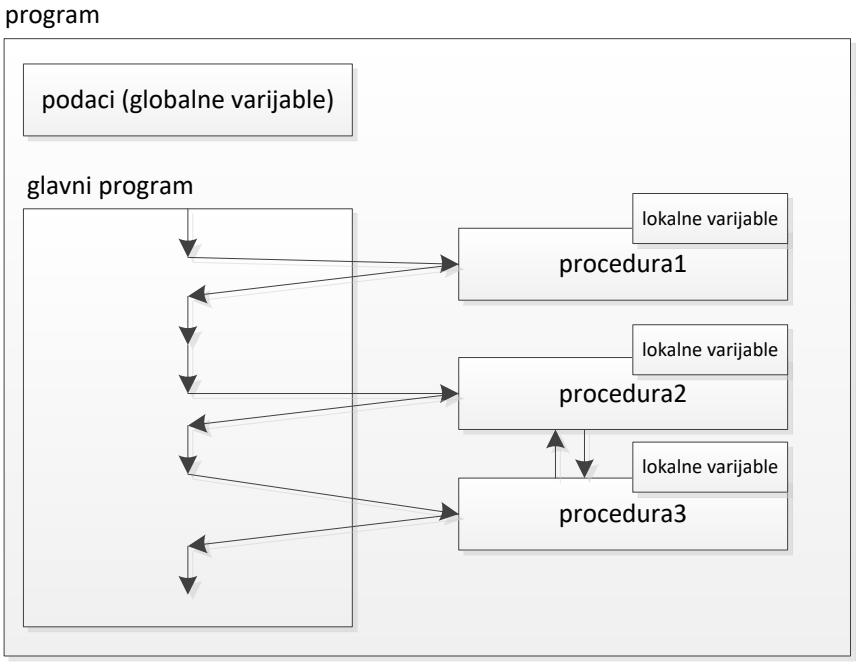
Glavni izazov u razvoju programske potpore predstavlja upravljanje kompleksnošću, koja raste s razvojem programskih tehnologija i očekivanjima korisnika na mogućnosti i funkcionalnost programske potpore. Kako bi se učinkovito i održivo moglo upravljati kompleksnošću u razvoju programske potpore, potrebno je prilagoditi ne samo pristupe upravljanju procesima programskog inženjerstva, već i paradigme na kojima se temelji sam proces programiranja.

Prva računala izvodila su za današnje pojmove razmjerno jednostavne programe koji su prvenstveno služili za ubrzavanje složenih matematičkih izračuna. Takvi programi mogli su obavljati osnovne računske operacije i upravljati jednostavnim mehanizmima kontrole programskog tijeka (uvjetno grananje, petlje i sl.). Instrukcije su se unosile na razini strojnog jezika, koji je prirodan način reprezentacije instrukcija računalu, ali ne i čovjeku. Programiranje je bilo dugo i mukotrpno, neintuitivno, sklono pogreškama, a bili su potrebni ponekad i tjedni da se naprave razmjerno jednostavni programi kodirani izravno u strojnom jeziku, pri čemu su prva računala mogla izvoditi svega nekoliko tisuća računskih operacija u sekundi. Takav pristup programiranju bio je *nestrukturiran*, a može ga se promatrati kao niz instrukcija koje čine jedan monolitni program bez dekompozicije problema u manje cjeline (pozive potprograma i procedura) i gdje se svi podaci tretiraju kao globalne varijable, kako je prikazano na slici 6.1.



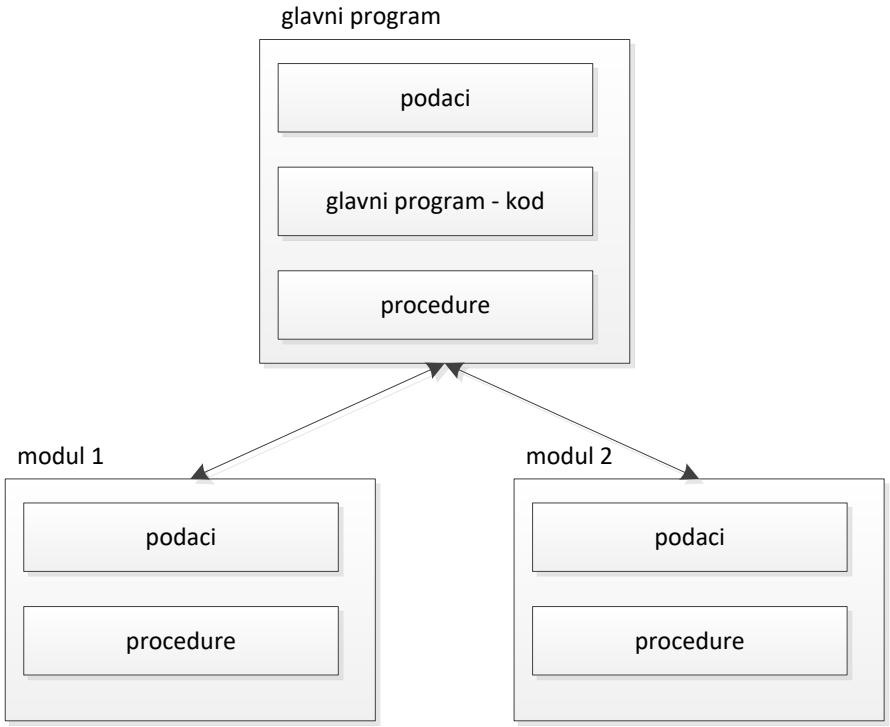
Slika 6.1 Nestrukturirana paradigma programiranja

Nestrukturirano programiranje koristilo se u samim počecima razvoja računala i informacijskih tehnologija, ali ubrzo su problemi koje su računala trebala rješavati postali bitno složeniji od pukog rješavanja matematičkih izraza. Pojavili su se programi s korisničkim sučeljem, bilo je potrebno upravljati brojnim ulazno-izlaznim jedinicama, ostvariti mrežno povezivanje i razmjenu različitih poruka, pružiti sigurnost i zaštitu informaciju, omogućiti istovremeni rad više korisnika, istovremeno izvođenje više programa, upravljati efikasno pohranom i dohvatom podataka itd. To je dovelo do potrebe za promjenom pristupa razvoju programske potpore na način da se program podijeli na manje cjeline koje rješavaju specifične podzadatke i gdje te programske cjeline mogu po potrebi pozivati jedna drugu. Takve cjeline nazivaju se *procedure*, a pristup razlaganju programa na manje specijalizirane programske cjeline *proceduralna paradigma*. Programiranje se pojavom takve paradigme podignulo na višu razinu apstrakcije i pojavili su se proceduralni viši programski jezici koji su omogućili da programer više ne razmišlja u okvirima građe i arhitekture računala, već da probleme rješava bliže izvornoj domeni problema. Prvi proceduralni jezici bili su COBOL, Fortran, Pascal, C i dr., a pristup proceduralne paradigme programiranja shematski je prikazan na slici 6.2.



Slika 6.2 Procedurala paradigma programiranja

Proceduralna paradigma značajno je olakšala razvoj složenih aplikacija pristupom rješavanja problema „podijeli-pa-vladaj“. Daljnja razrada tog pristupa dovodi do paradigme *modularnog programiranja*, u kojoj se programske procedure sa zajedničkom ili srodnom funkcionalnošću grupiraju u programske *module*, što znači da se program ne sastoji samo od jedne cjeline (modula), već od puno manjih zasebnih cjelina koje su u međusobnoj interakciji preko poziva procedura. Takav pristup potiče stvaranje ponovno iskoristivih programskih biblioteka (engl. *reusable libraries*) koje značajno ubrzavaju razvoj programske potpore jer je kôd određene funkcionalnosti potrebno napisati samo jednom, zapakirati u modul i nakon toga ga se može koristiti mnogo puta u novim programima. Pristup modularne paradigme programiranja prikazan je shematski na slici 6.3.



Slika 6.3: Modularna paradigma programiranja

Dugo vremena proceduralna i modularna paradigma predstavljale su dominantan pristup u razvoju programske potpore jer su omogućavale dovoljno robustan i skalabilan okvir za razvoj različitih vrsta programa. Proceduralna i modularna paradigma i danas se smatraju optimalnim izborom u nekim domenama, primjerice u razvoju jezgre operacijskih sustava, programske potpore niske razine (npr. upravljački programi za sklopovlje računala),

biblioteke specijalne namjene kod kojih su izuzetno važne performanse (matematički izračuni, kriptiranje, biblioteke za računalnu grafiku, zvuk i sl.), ugradbeni računalni sustavi itd.

Međutim, napretkom tehnologije, porastom očekivanja korisnika i pojavom novih domena u kojima je bilo potrebno razvijati sve složenija programska rješenja, proceduralna i modularna paradigma pokazale su se neučinkovitima i neodrživima u smislu mogućnosti upravljanja komplesnošću procesa programskog inženjerstva i implementacije rješenja koja su zahtijevala fleksibilnost, kako u samom procesu razvoja, tako i nadogradnje, zbog čestih interevencija i zahtjeva od strane krajnjih korisnika takvih sustava. Prvenstveno se to odnosi na područje poslovnih aplikacija, korisničkih programa za desktop i mobilne platforme, web aplikacije, računalne igre i sl. Temeljni problem proceduralne i modularne paradigme bio je u tome što je programer previše usmjeren na pitanje kako domenu problema iz stvarnog svijeta prevesti i opisati ograničenim konstruktima proceduralnog programskog jezika, umjesto da se fokusira na formuliranje programskog rješenja koje prirodno prati način razmišljanja i modeliranja rješenja kakvog koristimo u svakodnevnom životu kod rješavanja problema iz stvarnog svijeta. Proceduralna i modularna paradigma su povećanjem kompleksnosti programskih rješenja pokazale slične nedostatke kao i vodopadni model programskog inženjerstva pa je bilo potrebno razviti nove paradigme programiranja koje bi na bolji način upravljale složenim programskim rješenjima.

Objektno orijentirana paradigma (OOP) temelji se na konceptu *objekta* koji objedinjuje podatke i ponašanje te omogućuje modeliranje i programiranje na višem stupnju apstrakcije od proceduralnog programiranja jer nudi programsku okolinu u kojoj se o problemu promišlja na način blizak percepciji i razmišljanju ljudi prilikom rješavanja problema u stvarnom svijetu. Premda su se prvi programski jezici temeljeni na OOP paradigmi pojavili i razvijali još od šezdesetih godina prošlog stoljeća (Simula, Smalltalk, Ada itd.), veću popularnost OOP paradigma stječe tek devedesetih godina prošlog stoljeća, uz pojavu i popularizaciju brojnih programskih jezika koji podržavaju OOP paradigmu (C++, Java, Delphi, C#, Perl, Python, Ruby itd.).

Osnovni koncept na kojem se temelji OOP paradigma je pojam *objekta*, koji predstavlja reprezentaciju nekog entiteta iz stvarnog svijeta ili implementacije programskog rješenja, kojem se pridružuju *podaci* i *ponašanje*. Temeljna razlika u odnosnu na proceduralnu paradigmu je u tome što su u proceduralnoj paradigmi podaci striktno razdvojeni od procedura, dok ih koncept objekta objedinjuje u jednu cjelinu. Upravo su objekti s takvim svojstvima ključan element koji omogućuje modeliranje problema blisko načinu prirodnog razmišljanja ljudi prilikom rješavanja problema u stvarnom svijetu.

6.2 Razredi i objekti

Osnovu OOP paradigme čine usko povezani pojmovi objekta (engl. *object*) i razreda (engl. *class*). *Objekt* predstavlja reprezentaciju nekog entiteta iz vanjskog svijeta ili konceptualnog elementa unutar objektno-orijentirane programske okoline, a koji nakon stvaranja zauzima prostor u memoriji računala i pohranuje svoje trenutno stanje (podatkovni sadržaj značajki (atributa) objekta) te implementira funkcionalnost koju objekt može izvršiti nad podacima ili za neke programske entitete (ponašanje objekta). *Razred* ili *klasa* predstavlja apstraktni tip podataka koji služi kao predložak za stvaranje objekata u kojem su definirani atributi objekta (podatkovni sadržaj) i ponašanje objekta (opisano pomoću tzv. *metoda* ili članskih funkcija (engl. *member functions*)). Objekt se u smislu gornjih definicija može promatrati kao *instanca* ili *konkretni primjerak* razreda koji je stvoren u memoriji računala na temelju predloška sadržanog u definiciji klase, pri čemu svaki objekt ima vlastitu kopiju trenutnog stanja svih podatkovnih atributa.

Kako bi se u nastavku lakše prezentirali i objasnili koncepti OOP paradigme na konkretnim primjerima, koristit će se programski jezik C++. Zbog ograničenog prostora bit će prikazane samo osnovne mogućnosti koje pruža programski jezik C++ u okviru svoje specifične implementacije OOP paradigme.

Zamislimo primjer računalne simulacije ili igre u kojoj imamo automobile koji se utrkuju i kreću po zadanoj stazi. Premda bi se program koji simulira njihovo ponašanje mogao implementirati primjenom proceduralne i modularne paradigme, u nastavku će biti prikazano kako OOP paradigma omogućuje lakši i efikasniji razvoj sustava implementacijom koncepata na način koji je blizak ljudskom razmišljanju u prirodnoj domeni problema.

Na početku ćemo definirati objekt (klasu) tipa `Automobil`:

```
class Automobil
{
public:
      float kapacitetSpremnikaGoriva;
      float kolicinaGorivaUSpremniku;
      Vector2 polozajNaKarti;
      Vector2 trenutnaBrzina;
      bool jeLiUkljucen;

      void Ukljuci();
      void Iskljuci();
      void DodajGas(float iznos);
      void SmanjiGas(float iznos);
      void SkreciDesno(float kut);
      void SkreciLijevo(float kut);
      void Koci(float iznos);
      void NatociGorivo(float kolicina);
};
```

U prikazanom primjeru koristili smo ključnu riječ `class` programskog jezika C++ kako bismo definirali smo novi tip podatka `Automobil`. Ključna riječ `class` koristi se na sličan način kao ključna riječ `struct` u programskom jeziku C, koja omogućuje definiranje novih složenih tipova podataka (struktura), s tom razlikom da se u C-u dozvoljava da polja budu samo podaci, npr.:

```
struct Automobil
{
	float kapacitetSpremnikaGoriva;
	float kolicinaGorivaUSpremniku;
	Vector2 polozajNaKarti;
	Vector2 trenutnaBrzina;
	bool jeLiUkljucen;
};
```

Sve operacije nad automobilom bismo u programskom jeziku C trebali izvoditi pozivom procedura koje bi mijenjale unutarnje stanje instance strukture automobila. U takvom pristupu su podaci i procedure razdvojeni, što čini implementaciju programa složenijom i nije način na kojeg prirodno doživljavamo automobil u stvarnom svijetu. OOP pristup omogućuje da koncept automobila predstavimo objektom kojem su i sadržaj i ponašanje integrirani u jednu cjelinu.

Napomena: U programskom jeziku C++ ključna riječ `struct` ima nešto šire mogućnosti od onih koje ima u programskom jeziku C te je moguće u C++ strukture uključiti i članske funkcije. Oni koje zanima više mogu u specifikaciji programskog jezika C++ proučiti koje su razlike između ključnih riječi `struct` i `class`.

U glavnom programu instancirat ćemo dva automobila nad kojima ćemo izvršiti neke uobičajene radnje:

```
void main()
{
Automobil kiaSportage1, smartForTwo1;

	strcpy(kiaSportage1.naziv, "Kia Sportage");
	kiaSportage1.kapacitetSpremnikaGoriva = 62;
	kiaSportage1.kolicinaGorivaUSpremniku = 32;
	kiaSportage1.polozajNaKartiX = -5;
```

```
	kiaSportage1.polozajNaKartiY = 0;
	kiaSportage1.smjerX = 0;
	kiaSportage1.smjerY = 1;
	kiaSportage1.trenutnaBrzina = 0;
	kiaSportage1.jeLiUkljucen = 0;

	strcpy(smartForTwo1.naziv, "Smart Fortwo");
	smartForTwo1.kapacitetSpremnikaGoriva = 28;
	smartForTwo1.kolicinaGorivaUSpremniku = 10;
	smartForTwo1.polozajNaKartiX = 5;
	smartForTwo1.polozajNaKartiY = 0;
	smartForTwo1.smjerX = 0;
	smartForTwo1.smjerY = 1;
	smartForTwo1.trenutnaBrzina = 0;
	smartForTwo1.jeLiUkljucen = 0;

	kiaSportage1.NatociGorivo(30);
	kiaSportage1.Ukljuci();
	kiaSportage1.DodajGas(10);

	smartForTwo1.NatociGorivo(50);
	smartForTwo1.DodajGas(10);
printf("%s; %5.1f km/h, gorivo: %5.1f\n", kiaSportage1.naziv,
kiaSportage1.trenutnaBrzina, kiaSportage1.kolicinaGorivaUSpremniku);
	printf("%s; %5.1f km/h, gorivo: %5.1f\n", smartForTwo1.naziv,
smartForTwo1.trenutnaBrzina, smartForTwo1.kolicinaGorivaUSpremniku);
}
```

Primjer ispisa:

```
Kia Sportage;  40.0 km/h, gorivo: 62
Smart Fortwo:   0.0 km/h, gorivo: 28
```

U prikazanom primjeru smo na temelju istog predloška (klasa `Automobil`) instancirali dvije varijable istog tipa i nazvali ih `kiaSportage1` i `smartForTwo1`. Najprije smo za svaki objekt inicijalizirali početno stanje objekta postavljanjem vrijednosti u podatkovna polja (atributi, engl. *fields*), a zatim smo nad tim objektom pozvali neke funkcije koje opisuju ponašanje objekta (metode ili članske funkcije, engl. *methods*, *member functions*). Primjerice, za varijablu (objekt) `kiaSportage1` inicijalizirali smo sadržaj spremnika goriva na 32 litre, a zatim natočili gorivo za dodatnih 30 litara pozivom metode `NatociGorivo`, koju bismo mogli implementirati npr. na sljedeći način:

```
void Automobil::NatociGorivo(float kolicina)
{
      kolicinaGorivaUSpremniku += kolicina;
      if (kolicinaGorivaUSpremniku > kapacitetSpremnikaGoriva)
kolicinaGorivaUSpremniku = kapacitetSpremnikaGoriva;
}
```

Na ovom primjeru vidimo zašto je važan koncept objekta, koji uz podatke istovremeno sadrži i ponašanje implementirano članskim funkcijama: objekt preko članskih funkcija implementira ponašanje koje je njemu svojstveno, na mjestu koje je najprimjerenije, što je u definiciji objekta, a ne u nekoj vanjskoj proceduri koja nema izravne veze s objektom. Kada bismo ponašanje implementirali izvan objekta, onda bismo unaprijed morali znati pravilo npr. da ne možemo natočiti proizvoljnu količinu gorivo i trebali bismo toj proceduri također eksplicitno proslijediti informaciju o kapacitetu spremnika, što je objektu automatski dostupno putem atributa objekta.

U ispisu vidimo da smo za oba automobila natočili gorivo i dodali gas, s time da smo drugoga zaboravili uključiti. Članska funkcija `Ukljuci()` vodi računa da na ispravan način

inicijalizira unutarnje stanje da automobil ispravno radi (npr. postavljanjem zastavice `jeLiUkljucen` u stanje 1, uz dodatni posao oko inicijalizacije, kako će biti prikazano primjerom u nastavku), a članska funkcija `DodajGas()` mora znati da se brzina automobila ne može promijeniti ako automobil nije uključen. U ovom primjeru vidimo koliko koncept uključenja podataka i ponašanja u objekt olakšava programiranje i čini prirodnim opis problema iz domene stvarnog svijeta u programskom jeziku, što bi u slučaju proceduralne paradigme vodilo na raspršenu odgovornost između više struktura, procedura i modula, a time i puno teže upravljanje komplesnošću.

Objektno-orijentirana paradigma pruža mnogo više od jednostavnog povezivanja podataka i ponašanja u jednu cjelinu. Četiri su temeljna načela na kojima se zasniva puni potencijal OOP paradigme, a to su:

- apstrakcija (engl. *abstraction*),
- enkapsulacija (učahurivanje, engl. *encapsulation*),
- nasljeđivanje (engl. *inheritance*),
- polimorfizam (višeobličje, engl. *polymorphism*).

U nastavku će biti obrazložena sva četiri važna principa koja čine temelj OOP paradigme kroz praktične primjere.

6.3 Apstrakcija

Apstrakcija (engl. *abstraction*) predstavlja mogućnost modeliranja entiteta na način da se reperezentiraju samo *bitne* značajke stvarnog objekta. Primjerice, stvarni automobil može biti izuzetno složen ako se uzmu u obzir svi dijelovi od kojih je građen, kao i način na kojega ga se upravlja i fizikalne zakonitosti prema kojima se kreće. Računalni model automobila u OOP paradigmi predstavljen je objektom, koji sadrži samo one bitne značajke važne za kontekst programa u kojem se koristi model automobila. U tom smislu objekti ne teže potpuno vjernoj reprezentaciji stvarnih objekata u programskim strukturama, već apstrahiraju samo ono najvažnije, pri čemu formalni model apstrakcije podrazumijeva apstrahiranje podataka vezanih uz stanje objekta i ponašanja tog objekta, što je u OOP paradigmi predstavljeno članskim varijablama i funkcijama. U tom smislu klasa `Automobil`

predstavlja apstraktni model stvarnog automobila prikladan za reprezentaciju u računalnom programu jer zanemaruje sve što unosi nepotrebnu kompleksnost i koncentrira se isključivo na značajke koje su bitne.

Apstrakcija kao pojam povezana je usko i s pojmovima enkapsulacije i nasljeđivanja, koji će biti objašnjeni u nastavku. Apstrakcija se također odnosi i na definiranje minimalnog *sučelja* koje nam je važno za korištenje objekta, a u objektno-orijentiranoj paradigmi također kod modeliranja objekata iz domene problema možemo koristiti pristup da krenemo od apstraktnijih tipova podataka iz kojih izvodimo specifične.

6.4 Enkapsulacija

U primjeru klase `Automobil` prikazano je da klase (objekti) objedinjuju podatke i ponašanje u istoj programskoj cjelini kako bi se što prirodnije opisali entiteti iz domene problema. Zamislimo da funkcija za uključivanje automobila obavlja sljedeći posao:

```
void Automobil::Ukljuci()
{
bool pass;
      pass = ProvjeriJeLiIspravanKljuc();
      if (pass) pass = ProvjeriJeLiIspravnaRazinaUlja();
      if (pass) pass = ProvjeriJeLiIspravnaAirBag();
      ...
      if (!pass)
      {
            jeLiUkljucen = 0;
            return;
      }
      jeLiUkljucen = 1;
}
```

Pozivom u glavnoj funkciji

```
kiaSportage1.Ukljuci();
```

objekt automatski prema definiranom obrascu obavlja cijeli niz provjera prije nego postavi da je zastavica `jeLiUkljucen` aktivna i ona ne bi trebala biti postavljena ako neka od provjera nije zadovoljena. Međutim, s obzirom da se izvan klase mogu izravno dohvatiti polja s podacima, sasvim je moguće napraviti sljedeće:

```
strcpy(kiaSportage1.naziv, "Kia Sportage");
kiaSportage1.kapacitetSpremnikaGoriva = 62;
kiaSportage1.kolicinaGorivaUSpremniku = 32;
...
kiaSportage1.trenutnaBrzina = 0;
kiaSportage1.jeLiUkljucen = 0;

kiaSportage1.NatociGorivo(30);
// kiaSportage1.Ukljuci();
kiaSportage1.jeLiUkljucen = 1; // ukljucimo rucno zastavicu bez
provedenih provjera!
kiaSportage1.DodajGas(10);
```

Funkcija koja implementira dodavanje gasa obavlja provjeru zastavice `jeLiUkljucen` kao potvrdu ne samo da je automobil uključen, već da su i prilikom uključivanja vozila obavljene sve potrebne provjere. Problem u gornjem primjeru je da se zastavica `jeLiUkljucen` može promijeniti ručno mimo funkcije `Ukljuci()`, što će dovesti objekt u neregularno stanje jer će biti označeno da je automobil uključen, a da nije prošao potrebnu provjeru koju je trebao prilikom uključenja (npr. razina ulja, sigurnosni sustavi itd.). Dakle, prevelika sloboda i vidljivost unutarnjeg stanja objekta izvana nije nužno dobra stvar, jer se u ovom primjeru može nekontrolirano utjecati na unutarnje stanje objekta na neprikladan način i to je važno spriječiti.

Stoga je kao jedan od važnih koncepata na kojim počiva OOP paradigma prepoznato načelo *enkapsulacije* (engl. *encapsulation*), koja omogućuje da se zaštite detaljni unutarnje implementacije objekta (podatkovni i ponašajni) od mogućnosti da im se pristupa izravno izvan objekta. To se naziva još i *skrivanje implementacije* (engl. *implementation hiding*) kako bi se od korisnika klase sakrili unutarnji detalji implementacije jer mu nisu bitni za korištenje objekta, a njihovo izlaganje mogućnosti izravnog pristupa daje korisniku klase mogućnost zloupotrebe koja neće pridonijeti načelu smanjenja kompleksnosti programske potpore. Primjerice, korisniku automobila važno je znati kako upravljati automobilom (volan, papučica gasa, mjenjač), ali mu nije bitno poznavati sve detalje kako je automobil sastavljen i što se u njemu nalazi (dapače, dobro je da su ti detalji unutarnje implementacije dovoljno odvojeni od sučelja za upravljanje i mogućnosti da korisnik na svoju ruku bilo što mijenja). Objekt korisnik vidi i upravlja njegovim ponašanjem preko *sučelja* (engl. *interface*), a ponašanje objekt interno definira preko *implementacije* (engl. *implementation*). Razdvajanje sučelja i implementacije, uz skrivanje implementacije od korisnika objekta, naziva se *učahurivanje* ili *enkapsulacija*. Korisnik objektom upravlja preko slanja poruka objektu, što zapravo predstavlja pozivanje *članskih funkcija* ili *metoda*.

Uočimo da je u definiciji klase `Automobil` na početku korištena ključna riječ `public`:

```
class Automobil
{
public:
      float kapacitetSpremnikaGoriva;
      float kolicinaGorivaUSpremniku;
      ...
}
```

Sve što se nalazi iza ključne riječi `public` (atributi i članske funkcije) dostupni su izvan objekta. Međutim, ako bismo umjesti ključne riječi `public` napisali `private`:

```
class Automobil
{
private:
      float kapacitetSpremnikaGoriva;
      float kolicinaGorivaUSpremniku;
      Vector2 polozajNaKarti;
      Vector2 trenutnaBrzina;
      bool jeLiUkljucen;



      void Ukljuci();
      void Iskljuci();
      void DodajGas(float iznos);
      void SmanjiGas(float iznos);
      void SkreciDesno(float kut);
      void SkreciLijevo(float kut);
      void Koci(float iznos);
      void NatociGorivo(float kolicina);
};
```

tada sljedeći kôd ne bi radio:

```
void main()
{
Automobil kiaSportage1;
      kiaSportage1.kapacitetSpremnikaGoriva = 62;
      kiaSportage1.kolicinaGorivaUSpremniku = 32;
      kiaSportage1.polozajNaKartiX = -5;
...
```

Problem je u tome što ključna riječ `private` zabranjuje da se članskoj varijabli ili funkciji može pristupiti izvana. Sve što je označeno ključnom riječju `private` dostupno je isključivo unutar objekta.

Izmijenimo kôd na način da funkcije za upravljanje objektom budu dostupne izvana, ali da članske varijable ostanu sakrivene i da im korisnik klase ne može pristupiti izvana:

```
class Automobil
{
private:
      float kapacitetSpremnikaGoriva;
      float kolicinaGorivaUSpremniku;
      Vector2 polozajNaKarti;
      Vector2 trenutnaBrzina;
      bool jeLiUkljucen;

public:
      void Inicijaliziraj();
      void Ukljuci();
      void Iskljuci();
      void DodajGas(float iznos);
      void SmanjiGas(float iznos);
      void SkreciDesno(float kut);
      void SkreciLijevo(float kut);
      void Koci(float iznos);
      void NatociGorivo(float kolicina);
};
```

```
void Automobil::Inicijaliziraj()
{
	strcpy(kiaSportage1.naziv, "Kia Sportage");
	kiaSportage1.kapacitetSpremnikaGoriva = 62;
	kiaSportage1.kolicinaGorivaUSpremniku = 32;
	kiaSportage1.polozajNaKartiX = -5;
	kiaSportage1.polozajNaKartiY = 0;
	kiaSportage1.smjerX = 0;
	kiaSportage1.smjerY = 1;
	kiaSportage1.trenutnaBrzina = 0;
	kiaSportage1.jeLiUkljucen = 0;
}
```

U tom slučaju sljedeći kôd će raditi ispravno:

```
void main()
{
Automobil kiaSportage1;

	kiaSportage1.Inicijaliziraj();
	kiaSportage1.Ukljuci();
	kiaSportage1.DodajGas(10);
```

Uočimo da je ovaj kôd znatno jednostavniji od prvog primjera i da korisnik više ne mora voditi brigu o tome da inicijalizira objekt, ali i da više nema mogućnosti da korisnik klase objekt dovede u nedozvoljeno stanje izravnim mijenjanjem članskih varijabli koje predstavljaju unutarnje stanje objekta. Sada je odgovornost na inicijalizaciju prebačena s korisnika na objekt, a objekt je zaštićen od mogućnosti neopreznog dovođenja u nedozvoljeno stanje.

Inicijalizacija objekta vrlo je važna stvar i u programskom jeziku C++ postoji posebna funkcija koju prevoditelj zove odmah prilikom kreiranja objekta, a zove se *konstruktor* (engl. *constructor*). Konstruktor ima isto ime kao i naziv klase:

```
class Automobil
{
private:
      float kapacitetSpremnikaGoriva;
...
      bool jeLiUkljucen;


private:
      void Inicijaliziraj();
      void Ukljuci();
      void Iskljuci();
      ...
      void NatociGorivo(float kolicina);
};
void Automobil::Automobil()
{
Inicijaliziraj();
}
```

Glavni program sada možemo pisati ovako:

```
void main()
{
Automobil kiaSportage1;


      kiaSportage1.Ukljuci();
      kiaSportage1.DodajGas(10);
```

Ovdje više nije potrebno eksplicitno pozivati metodu za inicijalizaciju objekta jer će ju u gornjem primjeru automatski pozvati konstruktor u trenutku inicijalizacije. C++ također definira i posebnu funkciju koja se poziva prilikom uništavanja objekta, a zove se *destruktor* (engl. *destructor*), čija je prvenstvena namjena oslobađanje dinamički zauzetih memorijskih resursa kako bi se spriječio problem *curenja memorije* (engl. *memory leak*). S obzirom da je oslobađanje dinamički zauzetih memorijskih resursa potrebno ručno implementirati u destruktoru i da je izvor dodatne složenosti i mogućnosti pogrešaka, neki drugi OOP jezici (poput Jave, C# i sl.) koriste automatsko upravljanje memorijom (*memory-managed programming languages*) i oslobađaju programera potrebe za brigom o oslobađanju memorijskih resursa korištenjem tzv. *sakupljača smeća* (engl. *garbage collector*), koji automatski oslobađa zauzetu memoriju kada više nije referencirana.

Enkapsulacija omogućuje skrivanje dijela implementacije objekta kako bi se smanjila nepotrebna kompleksnost programa, postigla slabija povezanost među objektima kako bi objekti postali manje ovisni jedni o drugima i kako bi se time smanjila ukupna kompleksnost programa zatvaranjem specifičnosti unutarnje strukture i ponašanja klase u objekte. Preporučljivo je potpuno zaštititi članske varijable tako da se njihova vrijednost mijenja isključivo temeljem poziva članskih funkcija, koje posredno mijenjaju vrijednost članskih varijabli, u što se može uključiti i dodatna funkcionalnost u smislu provjere, pozadinskih efekata na druge varijable i sl. Primjerice, može se korisniku klase omogućiti da promijeni stanje privatne članske varijable ili pročita njezin sadržaj putem članskih funkcija:

```
class Automobil
{
private:
      float kapacitetSpremnikaGoriva;
      float kolicinaGorivaUSpremniku;

public:
      void SetKolicinaGorivaUSpremniku(float iznos);
      float GetKolicinaGorivaUSpremniku();
};

void Automobil::SetKolicinaGorivaUSpremniku(float iznos)
{
      kolicinaGorivaUSpremniku = iznos;
      if (kolicinaGorivaUSpremniku > kapacitetSpremnikaGoriva)
            kolicinaGorivaUSpremniku = kapacitetSpremnikaGoriva;
}

void Automobil::GetKolicinaGorivaUSpremniku()
{
      return kolicinaGorivaUSpremniku;
}
```

Funkcije koje postavljanju vrijednosti privatnih polja zovu se *setter* funkcije, a funkcije koje vraćaju vrijednosti *getter* funkcije. Nije nužno (niti preporučljivo) da sva polja imaju odgovarajuće parove *getter*/*setter* funkcija, neka polja mogu imati samo *getter* funkcije (*read-only* polja), a u praksi se često u *setter* funkcijama osim jednostavnog postavljanja vrijednosti polja rade i dodatne provjere ispravnosti postupka promjene vrijednosti polja.

6.5 Nasljeđivanje

Sljedeći važan pojam na kojem se temelji OOP paradigma je nasljeđivanje (engl. *inheritance*). U prvom primjeru imali smo dvije vrste automobila gdje smo za svakoga rekli da pripada istoj klasi `Automobil`, a onda smo ručno inicijalizirali vrijednosti polja koja su svojstvena svakom tipu automobila (npr. naziv modela, najveća brzina, kapacitet spremnika i sl.). OOP apstrakcija znači da smo u klasu `Automobil` uključili samo polja i ponašanja koja smatramo da su važna u modelu stvarnog automobila u našem programu, a kasnije smo kroz enkapsulaciju zaštitili polja objekta i inicijalizaciju prepustili funkciji `Inicijaliziraj()`, koju je automatski pozvao konstruktor prilikom kreiranja objekta. Na taj način smo olakšali korištenje klase u glavnom programu, ali smo izgubili fleksibilnost koju smo ranije imali kada su polja bila označena s `public` i nakon provedene enkapsulacije poljima više ne možemo izravno postavljati vrijednosti iz glavnog programa.

Jedno od mogućih rješenja bilo bi da se za sva polja od interesa (npr. naziv modela automobila, najveća brzina, veličina spremnika goriva vozila itd.) definiraju *getter*/*setter* funkcije i u glavnom programu pozivom *setter* funkcija postavimo različite vrijednosti tih polja. Međutim, u općem slučaju tim pristupom ne bismo suštinski iskoristili prednosti enkapsulacije i skrivanja detalja unutarnje implementacije objekta jer bismo morali za svaki tip automobila znati točan naziv modela i točan kapacitet spremnika goriva što bismo *svaki put* prilikom inicijalizacije automobila određenog tipa ručno trebali upisivati. Puno efikasniji pristup je da se apstrakcija nadopuni konceptom *nasljeđivanja* (engl. *inheritance*), koji kaže da iz klase koja predstavlja apstraktni predložak za instanciranje objekta možemo izvoditi (nasljeđivati) nove klase, koje sadrže (nasljeđuju) sve što smo definirali za baznu klasu (engl. *base class*), uvodeći pritom posebne specifičnosti vezane uz izvedenu klasu (engl. *derived class*). Neka je bazna klasa `Automobil` definirana na sljedeći način:

```
// base class
class Automobil
{
protected:
      char naziv[64];
```

```
    float kapacitetSpremnikaGoriva;
    float kolicinaGorivaUSpremniku;
    float polozajNaKartiX;
    float polozajNaKartiY;
    float smjerX;
    float smjerY;
    float trenutnaBrzina;
    float maksimalnaBrzina;
    bool jeLiUkljucen;

public:
    void Ukljuci();
    void Iskljuci();
    void DodajGas(float iznos);
    void SmanjiGas(float iznos);
    void SkreciDesno(float kut);
    void SkreciLijevo(float kut);
    void Koci(float iznos);
    void NatociGorivo(float kolicina);
};
```

Klasa `Automobil` predstavlja najapstraktniji model automobila, koji nije vezan uz specifičnog proizvođača i model. No u stvarnom svijetu nemamo samo jedan „generički“ model automobila, već ih imamo mnogo i to želimo na efikasan način modelirati u našem programu. Tu na scenu stupa pojam *nasljeđivanja*, koji nam omogućuje da iz najapstraktnije reprezentacije nekog entiteta iz stvarnog svijeta objektnim modelom (iz bazne klase) izvedemo specifične modele koji nasljeđuju svojstva i ponašanja, ali mogu uvesti i *neka nova* svojstva i ponašanja, odnosno *promijeniti* podrazumijevano ponašanje objekta nasljeđeno iz bazne klase.

Za primjer izvedimo iz bazne klase jednu nasljeđenu:

```
class KiaSportage_2_0_CRDi: Automobil
{
void DodajGas(float iznos) { ... }
void SmanjiGas(float iznos) { ... }
}

void KiaSportage_2_0_CRDi::KiaSportage_2_0_CRDi()
{
      strcpy(kiaSportage1.naziv, "Kia Sportage");
      kapacitetSpremnikaGoriva = 62.0f;
      maksimalnaBrzina = 201.0f;
}
```

Klasa `KiaSportage_2_0_CRDi` sadrži identična polja kao i bazna klasa `Automobil`, zato što je iza imena klase iza znaka ':' navedeno ime bazne klase (znak ':' može se promatrati kao „*nasljeđuje od*“). Stoga već definirane članske varijable i funkcije nije potrebno ponovo definirati u klasi `KiaSportage_2_0_CRDi`. Međutim, važno je uočiti da smo u klasi `Automobil` za članske varijable koje želimo enkapsulirati koristili ključnu riječ `protected` umjesto `private` kako smo imali u prethodnom primjeru. Naime, ključna riječ `private` je omogućavala skrivanje implementacije objekta od vanjskog svijeta, ali ta ključna riječ također skriva implementaciju klase i od nasljeđene klase! Stoga je potrebno za članske funkcije i varijable koje želimo vidjeti u nasljeđenoj klasi definirati da imaju vidljivost `protected`, što znači skrivanje implementacije objekta od vanjskog svijeta, ali ne i od nasljeđene klase. Ključne riječi `private`, `protected` i `public` predstavljaju tzv. *modifikatore vidljivosti*.

Uočimo također da smo za klasu `KiaSportage_2_0_CRDi` napisali novi konstruktor. Konstruktor služi za inicijalizaciju objekta pa je prirodno sve specifičnosti vezane uz

konkretni model automobila inicijalizirati u konstruktoru izvedene klase. Pritom je važno voditi računa da se i dalje poziva najprije konstruktor bazne klase, koji će napraviti generičku inicijalizaciju svih polja onako kako to najbolje zna napraviti bazna klasa, a nakon toga će specifičnosti inicijalizacije izvedene klase biti prepuštene konstruktoru izvedene klase. S obzirom da se incijalizacija specifične klase obavlja u konstruktoru izvedene klase, korisnik klase više ne mora voditi računa da to ručno radi u kôdu u glavnom programu jer sada specifična klasa zna kako izgraditi model konkretnog tipa automobila. To značajno pojednostavljuje korištenje klase jer je uklonjen teret s programera koji u glavnom programu treba napraviti samo sljedeće:

```
void main()
{
KiaSportage_2_0_CRDi kia1;

       kia1.Ukljuci();
       kia1.DodajGas(10);
       ...
}
```

Samim činom instanciranja objekta `kia1` automatski se pozivaju konstruktori bazne i izvedene klase, čime se ispravno inicijalizira objekt koji predstavlja model tog tipa vozila, što znači da programer više ne mora voditi računa u glavnom programu što napraviti da se ispravno inicijalizira objekt koji predstavlja neki model vozila. Nakog toga dovoljno je da uključi automobil i doda gas, što maksimalno pojednostavljuje implementaciju i čitljivost glavnog programa. Uočimo također da je u izvedenoj klasi `KiaSportage_2_0_CRDi` *ponovo definirana* funkcija `DodajGas()`: to znači da u glavnom programu poziv `kia1.DodajGas(10)` izvodi implementaciju metode koja je *specifična* za tip automobila implementiran klasom `KiaSportage_2_0_CRDi`, a ne poziva generičku fukciju `Automobil.DodajGas()`, što omogućuje da se u izvedenim klasama definiraju ponašanja drugačija od bazne klase.

Definirajmo sada nasljeđene klase za druge tipove automobila tako da za svaku od njih definiramo konstruktore izvedenih klasa i specifične implementacije funkcija za dodavanje gasa i kočenje (koje zapravo opisuju specifičnosti u razlikama npr. u motoru, prijenosu, kočnicama i ostalim elementima od kojih je građen automobil):

```
class HondaCR_V_2_0_iVTEC: Automobil
{
      void DodajGas(float iznos) { ... }
      void SmanjiGas(float iznos) { ... }
}

class Smart_Fortwo: Automobil
{
      void DodajGas(float iznos) { ... }
      void SmanjiGas(float iznos) { ... }
}

      void HondaCR_V_2_0_iVTEC::HondaCR_V_2_0_iVTEC()
{
      strcpy(kiaSportage1.naziv, "Honda CRV");
      kapacitetSpremnikaGoriva = 58.0f;
      maksimalnaBrzina = 190.0f;
}

void Smart_Fortwo::Smart_Fortwo()
{
      strcpy(kiaSportage1.naziv, "Smart Fortwo");
      kapacitetSpremnikaGoriva = 28.0f;
      maksimalnaBrzina = 155.0f;
}
```

Glavni program u kojem instanciramo i pokrećemo tri različita tipa automobila korištenjem nasljeđivanja izgleda ovako:

```
void main()
{
KiaSportage_2_0_CRDi kia1;
HondaCR_V_2_0_iVTEC honda1;
Smart_Fortwo smart1;

      kia1.Ukljuci();
      honda1.Ukljuci();
      smart1.Ukljuci();

      kia1.DodajGas(10);
      honda1.DodajGas(10);
      smart1.DodajGas(10);
      ...

}
```

Nasljeđivanje nam je omogućilo da se dobije maksimalno kompaktan i pregledan kôd kod kojeg se odmah vidi koji su tipovi automobila instancirani i gdje se inicijalizacija automobila obavlja odmah u trenutku instanciranja, a dinamika vozila (dodavanje gasa) implementira specifično za svaki tip vozila, pri čemu je implementacija primjenom načela enkapsulacije skrivena od korisnika jer mu nije bitna za upotrebu klase i smanjuje mogućnost nehotične pogreške nedozvoljenim utjecajem na unutarnje stanje objekta.

6.6 Polimorfizam

Kao što je u prethodnom primjeru prikazano, nasljeđivanje može značajno unaprijediti razumijevanje i duljinu kôda jer omogućuje da se iz bazne klase razvijaju specifičniji modeli. Nasljeđena klasa također može predstavljati baznu klasu za neku drugu klasu koja ju nasljeđuje, unoseći novu razinu razrade specifičnosti u odnosu na polaznu baznu klasu. To se zove *objektna hijerarhija*, jer postoji hijerarhijski odnos između baznih klasa i izvedenih koje ih nasljeđuju.

Međutim, u određenim situacijama želimo imati mogućnost da objekte koji pripadaju nekoj klasi razmatramo s gledišta *najapstraktnije* klase koja opisuje dani objekt (bazna klasa), a pri tome očekujemo od programa da automatski razriješi *specifičnosti* klase, tj. koju ispravnu metodu pozvati u objektnoj hijerarhiji. *Polimorfizam* nam omogućuje da klasama koje su nasljeđene iz bazne klase pristupamo preko istog sučelja kao da se radi o objektima najapstraktnije bazne klase, a program će automatski prepoznati o kojoj se specifičnoj klasi radi i tu instancu objekta tretirati sa svojim specifičnostima, bez potrebe da programer zna o kojoj se točno nasljeđenoj klasi radi u objektnoj hijerarhiji. Primjerice, za automobil nam nije bitno znati o kojem se točno modelu radi da bismo znali da bilo koji automobil podržava operaciju `Ukljuci()` i `DodajGas`(). U određenim situacijama korisno je imati mogućnost da instance objekata ne promatramo preko specifičnih nasljeđenih klasa, već preko zajedničkog sučelja bazne klase.

Razmotrimo primjer modeliranja utrke u kojoj sudjeluje ukupno osam automobila, koji su odabrani od tri moguća modela automobila. Instancirat ćemo naprije objekte koji pripadaju određenim klasama (objekti `kia1` i `kia2` klasi `KiaSportage_2_0_CRDi`, objekti `honda1`, `honda2` i `honda3` klasi `HondaCR_V_2_0_iVTEC` i objekti `smart1` i `smart2` klasi `Smart_Fortwo`).

Nadalje, želimo automobile koji sudjeluju u utrci promatrati kao instance klase `Automobil` jer unaprijed ne znamo koji će točno modeli automobila sudjelovati u utrci, a svi izvedeni modeli automobila ujedno pripadaju i baznoj klasi `Automobil`. Stoga je pogodno skup svih automobila koji sudjeluju u utrci opisati npr. poljem `automobili` (iako se obično u C++-u koriste neke fleksibilnije dinamičke strukture podataka, poput vektora, lista i sl.) koje je

sastavljeno od pokazivača na objekte klase `Automobil`. Svakom elementu polja `automobili` pridjeljujemo adresu instanciranog objekta, ali s obzirom da su članovi polja `automobili` pokazivači na `Automobil`, moramo koristiti *cast* operator da svakom elementu polja pridijelimo tipizirani pokazivač `(Automobil*)`. Inicijalizacija automobila za utrku prikazana je u sljedećem odsječku kôda:

```
void main()
{
KiaSportage_2_0_CRDi kia1, kia2;
HondaCR_V_2_0_iVTEC honda1, honda2, honda3;
Smart_Fortwo smart1, smart2;
Automobil* automobili[20];

      automobili[0] = (Automobil*)&kia1;
      automobili[1] = (Automobil*)&kia2;
      automobili[2] = (Automobil*)&honda1;
      automobili[3] = (Automobil*)&honda2;
      automobili[4] = (Automobil*)&honda3;
      automobili[5] = (Automobil*)&smart1;
      automobili[6] = (Automobil*)&smart2;
      ...
}
```

Važno je naglasiti da smo prilikom stavljanja pokazivača na objekt u polje `automobili` *izgubili* informaciju o kojoj se izvedenoj klasi objekta radi jer polje pamti tipizirane pokazivače na `Automobil`. Dakle, ako promotrimo objekt `automobili[0]`, možemo reći da pripada klasi `Automobil`, ali nemamo informaciju da se zapravo radi o pokazivaču koji pokazuje na objekt klase `KiaSportage_2_0_CRDi` u memoriji.

Kada bismo u glavnom programu htjeli pokrenuti utrku svih prijavljenih automobila koje smo dodali u polje `automobili`, trebali bismo svaki od automobila upaliti i dodati mu gas. Postupak pokretanja utrke želimo promatrati na način da ne ovisimo o tome da moramo nužno unaprijed znati koji će se modeli automobila prijaviti i kako svaki od njih pokrenuti i dodati mu gas, već želimo pozvati iste generičke procedure nad svim automobilima i prepustiti programu da sam zaključi o kojem se modelu radi i za njega pokrene specifičnu proceduru paljenja i dodavanja gasa. Na taj način imamo najveću razinu apstrakcije problema, što se može prikazati sljedećim odsječkom kôda:

```
void main()
{
KiaSportage_2_0_CRDi kia1, kia2;
HondaCR_V_2_0_iVTEC honda1, honda2, honda3;
Smart_Fortwo smart1, smart2;
Automobil* automobili[20];
      automobili[0] = (Automobil*)&kia1;
      automobili[1] = (Automobil*)&kia2;
      automobili[2] = (Automobil*)&honda1;
      automobili[3] = (Automobil*)&honda2;
      automobili[4] = (Automobil*)&honda3;
      automobili[5] = (Automobil*)&smart1;
      automobili[6] = (Automobil*)&smart2;

      for(int i = 0; i < 7; i++)
      {
            automobili[i]->Ukljuci();
            automobili[i]->DodajGas(10);
      }
}
```

Dakle, postupak pokretanja utrke svodi se na iteriranje kroz polje `automobili` pomoću `for` petlje gdje se svaki automobil uključuje i dodaje mu se gas. Međutim, problem kod ovog pristupa je u tome što će pozivi funkcija

```
automobili[0]->Ukljuci();
automobili[0]->DodajGas(10);
```

pozvati članske funkcije bazne klase

```
Automobil::Ukljuci()
Automobil::DodajGas(float iznos)
```

što znači da će se svi automobili neovisno o modelu pokrenuti na isti „generički“ način kako je opisano u baznoj klasi, a neće se u obzir uzeti specifičnosti svakog modela prilikom njegovog uključenja i pokretanja. To nije željeno ponašanje jer zapravo i ne postoji „generički“ način koji opisuje kako uključiti i pokrenuti bilo koji proizvoljni model automobila. Opisano ponašanje programa je posljedica činjenice da u polju `automobili` imamo pokazivače na `Automobil` i na tom mjestu u programu nemamo informaciju da je element polja `automobili[0]` zapravo instanca nasljeđene klase `KiaSportage_2_0_CRDi`.

Da bismo postigli željeno ponašanje, potrebno je unijeti neke izmjene u baznu klasu `Automobil`:

```
// base class
class Automobil
{
protected:
      char naziv[64];
      float kapacitetSpremnikaGoriva;
      float kolicinaGorivaUSpremniku;
      float polozajNaKartiX;
      float polozajNaKartiY;
```

```
    float smjerX;
    float smjerY;
    float trenutnaBrzina;
    float maksimalnaBrzina;
    bool jeLiUkljucen;

public:
    virtual void Ukljuci();
    virtual void Iskljuci();
    virtual void DodajGas(float iznos);
    virtual void SmanjiGas(float iznos);
    virtual void SkreciDesno(float kut);
    virtual void SkreciLijevo(float kut);
    virtual void Koci(float iznos);
    virtual void NatociGorivo(float kolicina);
};
```

Ispred imena članskih fukcija dodali smo sada novu ključnu riječ `virtual`. Ključna riječ `virtual` upućuje program da kada koristimo referencu (pokazivač) na neki objekt prilikom poziva funkcije označene kao `virtual` vodi računa o tome da ne poziva varijantu dotične članske funkcije iz bazne klase, već da pozove funkciju koja pripada nasljeđenoj klasi. Uz izmijenjeni potpis funkcije u novoj inačici bazne klase `Automobil` nije potrebno naglašavati u nasljeđenoj klasi da se radi o virtualnoj funkciji:

```
class KiaSportage_2_0_CRDi: Automobil
{
void Ukljuci() { ... }
void DodajGas(float iznos) { ... }
}
```

Nakon što smo u baznoj klasi funkcije označili kao virtualne, u glavnom programu će pozivi funkcija za objekt `automobili[0]`

```
automobili[0]->Ukljuci();
automobili[0]->DodajGas(10);
```

rezultirati pozivom članskih funkcija iz nasljeđene klase:

```
KiaSportage_2_0_CRDi::Ukljuci()
KiaSportage_2_0_CRDi::DodajGas(float iznos)
```

Opisani mehanizam zove se *višeobličje* ili *polimorfizam*, jer omogućuje da se objekti promatraju kao instance bazne klase (npr. `Automobil`), a da se za vrijeme izvođenja programa (*run-time*) putem mehanizma virtualnih funkcija pozove ispravna verzija specifičnog ponašanja definiranog u članskoj funkciji nasljeđene klase, bez potrebe da poznajemo o kojoj se nasljeđenoj klasi radi jer će to za nas automatski zaključiti program. Polimorfizam je izuzetno koristan i moćan koncept koji omogućuje još apstraktniji način reprezentacije domene problema u OOP paradigmi i pridonosi boljem upravljanju komplesnošću razvoja programske potpore.

7 UML modeliranje

7.1 Modeliranje programske potpore pomoću UML-a

Proces programskog inženjerstva obuhvaća više faza razvoja programskog proizvoda, pri čemu izuzetno važnu ulogu imaju faze planiranja i analize zahtjeva. Konačni uspjeh razvoja složenog programskog proizvoda uvelike ovisi o tome koliko su zahtjevi kvalitetno prikupljeni i koliko dobro međusobno komuniciraju različiti dionici u razvoju programske potpore (naručitelji, razvojni tim, voditelji projekata, korisnici itd.). Agilni pristupi procesu programskog inženjerstva dodatno potiču inkrementalni i iterativni razvoj, s velikim naglaskom na uključenost korisnika i njegovu kontinuiranu povratnu informaciju za vrijeme procesa razvoja programskog proizvoda.

Projektiranje programske potpore predstavlja implementaciju zamišljenog modela programskog produkta u odabranom programskom jeziku i tehnologiji. Kako bi različiti dionici u razvoju programskog proizvoda mogli sudjelovati u definiranju i razradi modela programske potpore te praćenju i upravljanju napretkom razvoja proizvoda, potrebno je da svi koriste ujednačeni jezik i terminologiju, bez obzira na njihovu razinu poznavanja programskih tehnologija. Usporedo s razvojem složenosti programske potpore pojavio se problem kako učinkovito upravljati ne procesom programskog inženjerstva u uvjetima nepotpunih i kontinuirano promjenjivih zahtjeva, već i kako na unificiran način reprezentirati zahtjeve, strukturu i ponašanje programskih artefakata unutar kompleksnog programskog rješenja. Klasični vodopadni model programskog inženjerstva, koji se pokazao uspješnim u tradicionalnim industrijama, u velikoj mjeri napušten je i zamijenjen drugim pristupima koji su prikladniji u domeni razvoja programske potpore, dok je sličan odgovor bio potreban i u području modeliranja zahtjeva i opisa implementacije programskog rješenja.

Nekada su se programski razvojni inženjeri koristili *ad hoc* načinom opisivanje zahtjeva i implementacije programskog rješenja, koji su uključivali razne skice, dijagrame tijeka, tablice, tekstualne opise i sl., koji nisu bili strukturirani na jedinstven način, niti su bili ujednačeni u terminologiji i notaciji. Kao odgovor na potrebu za ujednačavanjem terminologije i notacije nastao je *Unified Modeling Language* (UML) kao standardizirani

jezik za specifikaciju, vizualizaciju, projektiranje i dokumentiranje artefakata programskih sustava. Prva verzija standarda UML v1.0 objavljena je 1997. godine od strane konzorcija Object Management Group (OMG), neprofitne organzacije koja se bavi različitim standardima vezanima uz modeliranje, primarno u području razvoja softvera i srodnih industrija.

UML se u velikoj mjeri oslanja na grafičku notaciju (dijagrame) u opisu zahtjeva, strukture i ponašanja programske potpore. UML nije usko vezan uz neki specifični programski jezik ili ograničen na određenu paradigmu razvoja programske potpore te se radi o okviru opće namjene koji je primjenjiv na modeliranje najrazličitijih programskih sustava. Premda je izvorno zamišljen za modeliranje sustava u području programskog inženjerstva, njegove primjene moguće su i u drugim domenama (npr. modeliranju poslovnih procesa, proizvodnje, prometne infrastrukture i sl.).

Premda UML nije ograničen na posebnu programsku paradigmu, objektno-orijentirana paradigma (OOP) uvelike je utjecala na motivaciju i razvoj UML standarda jer se pojavila praktična potreba za učinkovitim prikazom složenih OOP modela koji su trebali biti razumljivi i osobama bez tehničkih znanja iz područja programiranja. Jedan od važnih ciljeva UML notacije bio je upravo približiti i opisati složene programske koncepte na jednostavan način i osobama koje nisu programeri. UML se stoga može definirati i kao jednostavan mehanizam za modeliranje gotovo svih kompleksnih sustava koji se danas susreću u praksi.

UML tipično koristi *dijagrame* za modeliranje jer je vizualna reprezentacija sažetija i razumljivija od tekstualne. Postoji devet standardnih tipova UML dijagrama, koji su podijeljeni u dvije osnovne skupine, koje čine dijagrami za *strukturno* (statičko) i *ponašajno* (dinamičko) modeliranje sustava. UML dijagrami za strukturno modeliranje (engl. *structural modeling*) su sljedeći:

- dijagram razreda (engl. *class diagram*),
- dijagram objekata (engl. o*bject diagram*),
- dijagram komponenata (engl. c*omponent diagram*),
- dijagram razmještaja (engl. *deployment diagram*).

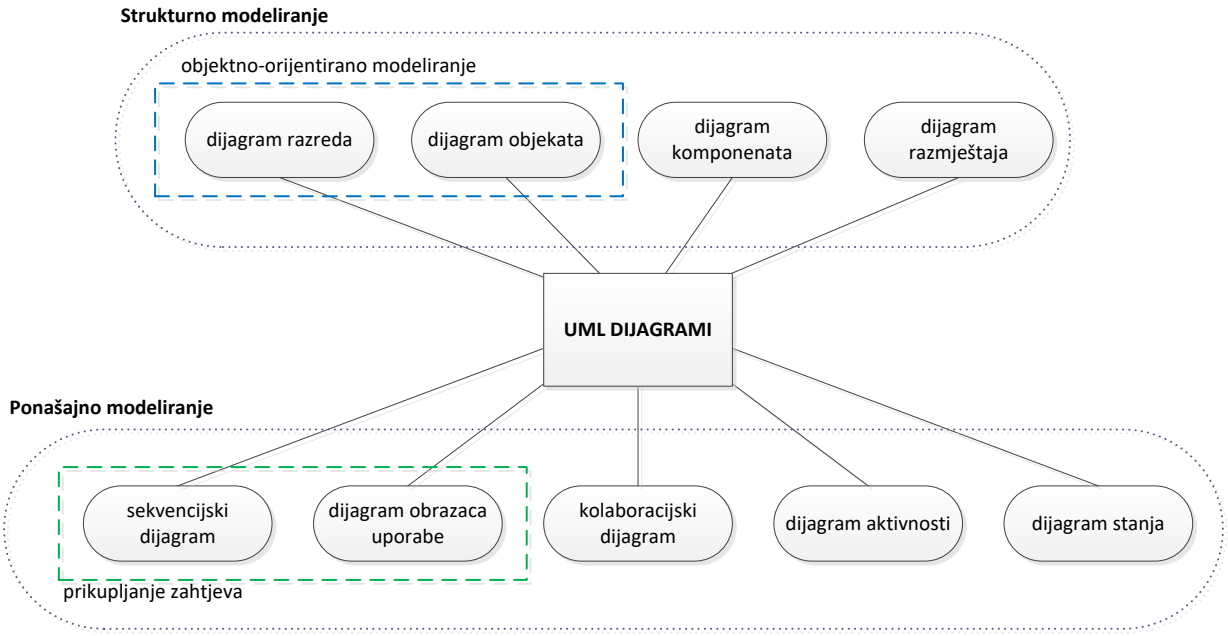
UML dijagrami za ponašajno modeliranje (engl. *behavioural modeling*) obuhvaćaju:

- dijagram obrazaca uporabe (engl. *use case diagram*),
- sekvencijski dijagram (engl. *sequence diagram*),
- kolaboracijski dijagram (engl. *collaboration diagram*),
- dijagram stanja (engl. *statechart diagram*),
- dijagram aktivnosti (engl. *activity diagram*).

Strukturno modeliranje predstavlja okvir za opisivanje građe i međusobnog odnosa entiteta unutar sustava, dok ponašajno modeliranje opisuje interakcije unutar sustava između entiteta opisanih strukturnim dijagramima.

Svaki UML dijagram predstavlja djelomičan opis sustava, odnosno pojednostavljeni model sustava promatran iz određene *perspektive*. Dijagrami su komplementarni i služe za modeliranje različitih *pogleda* na sustav. Prilikom modeliranja sustava nije nužno koristiti sve dijagrame, a sadržaj i razina detalja treba biti usklađena s potrebama i ciljevima, kako bi sustav bio dovoljno jasno i jednoznačno opisan, bez opterećivanja nepotrebnim detaljima.

Na slici 7.1 prikazani su standardni UML dijagrami. Za strukturiranje programske potpore temeljene na OOP paradigmi posebno su važni dijagrami razreda i objekata, a sekvencijski dijagrami i dijagrami obrazaca uporabe koriste se najviše u specifikaciji zahtjeva za razvoj programske potpore.



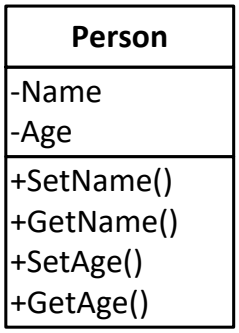
Slika 7.1: Vrste UML dijagrama

U nastavku će biti pobliže objašnjene sve vrste UML dijagrama na jednostavnim primjerima, uz tipične slučajeve kada se koriste u praksi.

7.2 Dijagram razreda (engl. *class diagram*)

Dijagram razreda (engl. *class diagram*) najčešće je korišteni tip dijagrama u UML notaciji. Ujedno je i jedan od najvažnijih u procesu programskog inženjerstva jer tipično služi za izgradnju modela implementacije programske potpore (u OOP paradigmi), pri čemu drugi pogledi često služe kao pomoć u izgradnji dijagrama razreda (npr. dijagram obrazaca uporabe opisuje funkcionalnosti, sekvencijski i kolaboracijski interakciju između objekata i sl.).

Dijagrami razreda opisuju atribute i operacije klasa te su pogodni za izravno mapiranje u entitete objektno-orijentiranih programskih jezika. Dijagrami razreda prikazuju klase, sučelja, asocijacije, međuovisnosti, ograničenja itd. Na slici 7.2 prikazan je primjer UML dijagrama razreda koji opisuje jednostavnu klasu *Person*.



Slika 7.2: UML dijagram razreda

Klasa se u dijagramu razreda prikazuje pravokutnikom, koji je podijeljen u tri manja u kojima se nalaze sljedeće informacije:

- naziv razreda (u gornjem dijelu)
- atributi – podatkovni elementi razreda (tj. članske varijable klase)
- operacije – ponašanje razreda (tj. članske funkcije klase)

U primjeru na slici 7.2 može se uočiti da se ispred naziva atributa i operacija nalaze prefiksi „+“ i „-“. Oni označavaju modifikatore vidljivosti članskih varijabli i funkcija, pri čemu „+“ označava *public*, „#“*protected*, a „-“ *private* modifikator vidljivosti. UML definicija razreda može sadržavati i druge informacije, kao što su tipovi podataka, ograničenja na vrijednosti koje se mogu dodijeliti varijablama, podrazumijevane (*default*) vrijednosti varijabli, parametre, povratne vrijednosti članskih funkcija itd.

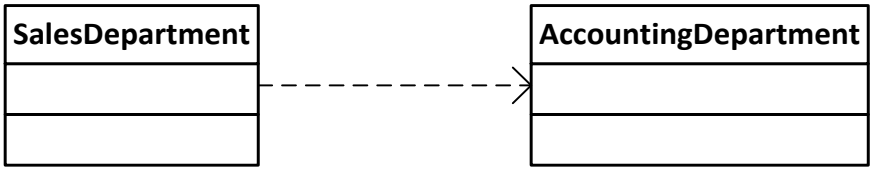
U OOP paradigmi programska potpora izgrađuje se na temelju skupa razreda koji služe za implementaciju željene funkcionalnosti produkta. Razredi su međusobnim odnosima koji se naznačuju dogovorenom notacijom na UML dijagramima, pri čemu se ti odnosi mogu podijeliti u dvije skupine:

- veze (engl. *relationship*) - apstraktni prikaz svih mogućih veza i međudjelovanja između instanci dvaju razreda, što obuhvaća međuovisnost klasa (engl. *dependency*), asocijacije između klasa (engl. *association*), sadržavanje jednog razreda unutar drugog (agregacija (engl. *aggregation*) i kompozicija (engl. *composition*)), višestrukost asocijacije (engl. *association multiplicity*) itd.

- generalizacija (engl. *generalization*) - opis organizacije objektne hijerarhije temeljene na načelima nasljeđivanja i polimorfizma, koji su svojstveni OOP paradigmi, pri čemu je potrebno naglasiti da generalizacija ne opisuje veze između razreda na način kako to radi asocijacija, nego definira odnose između njih u smislu nasljeđivanja klasa.

Primjeri korištenja veza i generalizacije u UML dijagramima razreda bit će opisani u nastavku.

Najlabaviju vezu među objektima čini veza *međuovisnosti* (engl. *dependency*). U UML terminologiji *dependency* znači da jedna klasa na neki način koristi drugu i ima informaciju o njezinom postojanju, ali ne postoji neka čvršća veza između tih klasa i njihova suradnja je privremena i tranzijentna.



Slika 7.3: Označavanje veze tipa *dependency* između klasa

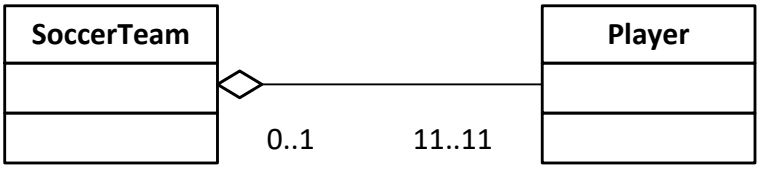
U primjeru na slici 7.3 vidi se da postoji veza između klasa *SalesDeparment* i *AccountingDeparment* što upućuje na činjenicu da između tih dviju klasa postoji povremena suradnja (npr. kada se u odjelu prodaje proda neki proizvod, informacija o tome šalje se u odjel računovodstva, ali te dvije klase osim te kratkotrajne komunikacije nakog tog događaja nisu u nekoj čvršćoj vezi u smislu da jedna drugu sadržavaju ili da su izvedene iz iste bazne klase). Međuovisnost se obično interpretira kroz izjavu korištenja („*Object A* ***uses an*** *object B*“).

Čvršći tip veze između dviju klasa predstavlja *asocijacija* (engl. *association*). Na slici 7.4 prikazana je asocijacija između klasa *Student* i *Faculty*. Prikazana veza upućuje da su studenti i fakultet povezani cijelo vrijeme, pri čemu je u prikaz dodatno uključena i *višestrukost asocijacije* (engl. *multiplicity*) oznakama iznad linije koja upućuje na asocijaciju. Znak „*“ označava da se na tom dijelu asocijacije može nalaziti proizvoljan broj objekata (studenata), a na dijelu asocijacije gdje se nalazi „1“ znači da se tu može nalaziti samo jedan

objekt (svi studenti su upisani samo na jedan fakultet). Relacija na slici 7.4 može se tumačiti kao izjava „da fakultet može pohađati proizvoljan broj studenata, ali da svaki student može pohađati samo jedan fakultet“. Asocijacija se obično interpretira kroz izjavu posjedovanja („*Object A* ***has an*** *object B*“).

Slika 7.4: Označavanje veze tipa *association* između klasa

Čvršća varijanta asocijacije među klasama predstavlja se *agregacijom* ili sadržavanjem (engl. *aggregation*). Agregracija predstavlja odnos u kojem se klase spajaju u novi složeniji objekt koji čini jednu cjelinu, ali na način da se objekti koji su instance tih klasa mogu razdvojiti i po potrebi egzistirati neovisno. Na slici 7.5 prikazana je agregacija između klasa *SoccerTeam* i *Player*. Nogometni tim sastoji se od igrača, koji također mogu egzistirati neovisno o timu. Na slici se također može uočiti višestrukost veze koja je definirana na način da neki igrač može biti u jednom ili nijednom timu (oznakom „0..1“), a da se tim sastoji od točno 11 igrača (oznaka „11..11“). Uočiti da poveznica između klasa među kojima postoji odnos agregacije ima mali romb na strani one klase koja sadržava drugu klasu. Agregacija se obično interpretira kroz izjavu vlasništva („*Object A* ***owns*** *object B*“).



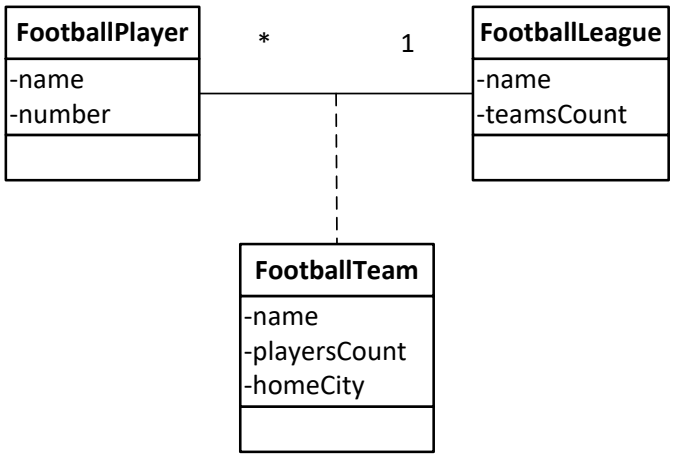
Slika 7.5: Označavanje veze tipa *aggregation* između klasa

Još čvršću varijantu asocijacije među klasama od agregacije predstavlja tzv. *kompozicija* (engl. *composition*). Slično kao agregacija, i kompozicija predstavlja odnos u kojem se klase spajaju u novi složeniji objekt koji čini jednu cjelinu, ali se u ovom slučaju radi o vrlo čvrstoj vezi među objektima, do točke da se ti objekti ne mogu rastaviti i egzistirati neovisno, tj. čine smislenu cjelinu samo ako jedan sadržava drugog. Ako se instanca klase koja sadržava drugu

uništi, automatski se uništava i sadržana klasa. Na slici 7.6 prikazana je kompozicija između klasa *House* i *Room.* Jedna kuća sastoji se od više prostorija, ali prostorije ne mogu postojati neovisno od kuće čiji su sastavni dio. Poveznica između klasa među kojima postoji odnos kompozicije slična je kao i kod agregacije, ali s razlikom da se koristi oznaka malog romba koja je ispunjena. Kompozicija se obično interpretira kroz izjavu pripadanja kao sastavnog dijela cjeline („*Object B* ***is a part of*** *object A*“).

Slika 7.6: Označavanje veze tipa *composition* između klasa

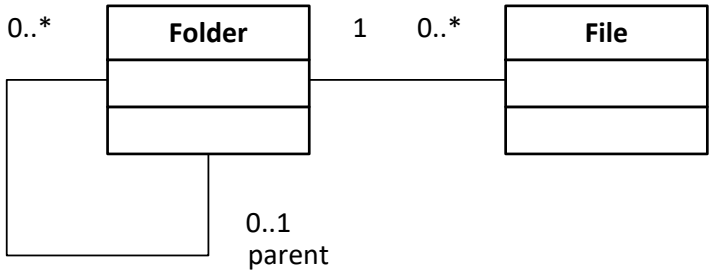
Postoje slučajevi kada se veze između elemenata dvije klase ne mogu opisati jednostavnom strukturnom poveznicom, već je odnos među njima kompleksniji i samu vezu između dvije klase treba modelirati dodavanjem nove klase koja će opisivati tu vezu. Primjer takvog složenijeg odnosa prikazan je na slici 7.7. U tom primjeru imamo klasu igrača (klasa *FootballPlayer*) koji igra u nogometnoj ligi (klasa *FootballLeague*), ali igrača ne možemo izravno povezati s ligom jer je svaki igrač u ligu uključen kroz pripadnost klubu (klasa *FootballTeam*). Igrač je istovremeno i član kluba i lige, ali se veza između igrača i lige ostvaruje posredno, putem pripadnosti klubu. Kada postoji takva složena povezanost dviju klasa, ona se modelira umjesto jednostavnom poveznicom tzv. *pridruženim razredom* (engl. *association class*). U ovom slučaju pridruženi razred je *FootballTeam* i značajke veze svakog igrača s ligom se modeliraju putem njega. Pridruženi razred prikazuje se kao normalna klasa, s tim da je grafički prikaz takav da se taj razred iscrtkanom linijom povezuje s poveznicom između dviju klasa koju pobliže dodatno opisuje.



Slika 7.7: Primjer korištenja pridruženog razreda za opis složenih veza među objektima

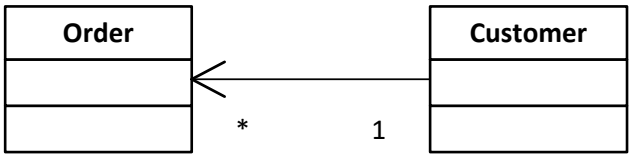
Postoji više mogućnosti kako relaciju opisanu pridruženim razredom implementirati u kodu. Uobičajeno je da se koriste tri klase, po jedna na svakom kraju poveznice (*FootballPlayer* i *FootballLeague* u ovom slučaju) i pridruženi razred (*FootballTeam* u ovom primjeru). Ovisno o implementaciji, može i ne mora postojati izravna veza između klasa na krajevima poveznice, odnosno implementacija može zahtijevati da se strogo razdvoje klase na krajevima poveznice i referenciraju isključivo kroz pridruženi razred.

Refleksivno pridruživanje (engl. *reflexive association*) predstavlja slučaj kada klasa upućuje na vezu sa samom sobom, tj. kada postoje veze između objekata koji predstavljaju instance iste klase. Na slici 7.8 prikazan je model datotečnog sustava u kojem postoje klase *Folder* i *File*. Svaka instanca klase *Folder* predstavlja objekt koji može imati 0 ili 1 roditelja (0 samo u slučaju da se radi o korijenskom (*root*) direktoriju), a svaki objekt *Folder* može sadržavati 0 ili proizvoljan broj poddirektorija. Opisana relacija predstavlja refleksivno pridruživanje jer se opisuje veza između objekata iste klase, a tipičan primjer refleksivnog pridruživanja je modeliranje stabla hijerarhije (npr. organizacijska struktura tvrtke, katalog web trgovine i sl.). Osim refleksivnog pridruživanja na slici 7.8, postoji i obična asocijacija koja kaže da svaki direktorij može sadržavati 0 ili proizvoljan broj datoteka.



Slika 7.8: Refleksivno pridruživanje na primjeru modela datotečnog sustava

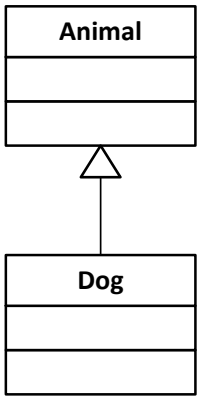
U pravilu je smjer pridruživanja asocijacija između klasa dvosmjeran (engl. *bidirectional*), osim ako se posebno ne naglasi jednosmjernost asocijacije (engl. *unidirectional*). Dvosmjerne asocijacije nemaju smjer pridruživanja sadržan u prikazu veze, dok je kod jednosmjerne asocijacije to bitno naglasiti. Na slici 7.9 prikazan je primjer dviju klasa, kupca (*Customer*) i narudžbe (*Order*). Premda bi se načelno asocijacija između kupca i narudžbe mogla modelirati i dvosmjerno, prirodnije je i bolje eksplicitno naglasiti da se radi o jednosmjernoj asocijaciji, pri čemu je važno navesti i smjer asocijacije. U prikazanom primjeru za odabranog kupca znaju se sve narudžbe pa stoga nema potrebe da se redundantno čuva informacija kako od narudžbe doći do kupca. U praksi je takva redundancija posebno nepogodna kod jezika s automatskim upravljanjem memorijom (tzv. *managed* jezici, kao što su npr. Java i C#), koji automatski mogu osloboditi memoriju objekata koji više nisu referencirani. U slučaju da postoje aktivne nepotrebne reference na objekte u memoriji, neće doći do oslobađanja memorije za one objekte koji se više ne koriste.



Slika 7.9: Primjer jednosmjerne asocijacije

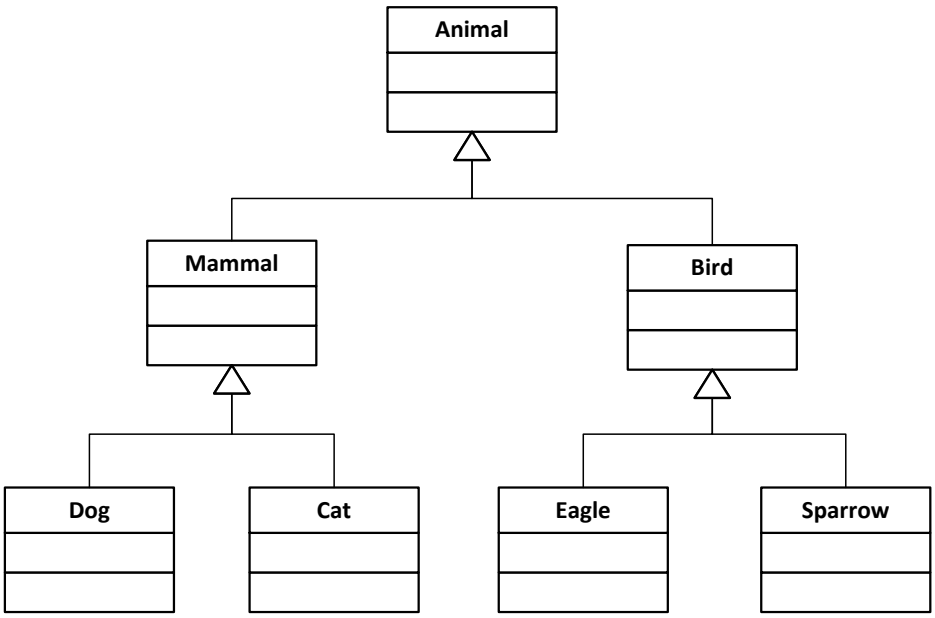
Sljedeća vrsta mogućeg odnosa među klasama odnosi se na njihov međusobni položaj u hijerarhiji nasljeđivanja. Generalizacija (engl. *generalization*) upućuje na odnos između bazne i nasljeđene klase, gdje nasljeđena klasa u UML dijagramu uvijek pokazuje na općenitiju baznu klasu iz koje se izvodi i čija svojstva nasljeđuje. Primjer generalizacije

prikazan je na slici 7.10: iz bazne klase *Animal* izvodi se nasljeđena klasa *Dog*, pri čemu je poveznica između bazne i nasljeđene klase prikazana strelicom, gdje strelica pokazuje od nasljeđene (specifičnije) prema baznoj (općenitijoj) klasi. Generalizacija se obično interpretira kroz izjavu istovjetnosti pripadanja uzvedene klase vrsti (tipu) klase iz koje se izvodi („*Object A* ***is an*** *object B*“, npr. „*pas* ***je*** *životinja*“).



Slika 7.10: Prikaz odnosa generalizacije (nasljeđivanja) između klasa

Prikaz hijerarhije klasa važan je za razumijevanje strukture i organizacije programa razvijenog prema OOP paradigmi. UML dijagrami razreda prikladni su za prikaz hijerarhije klasa, koja u stvarnim programima može biti vrlo složena. Primjer hijerarhije klasa prikazan je slici 7.11. Na slici može se vidjeti da se iz bazne klase *Animal* izvode klase *Mammal* i *Bird*, iz kojih se dalje izvode *Dog* i *Cat* (klase životinja koje pripadaju sisavcima (*Mammal*)) te *Eagle* i *Sparrow* (klase životinja koje pripadaju pticama (*Bird*)). Iz hijerarhije klasa vidljivo je da klase *Dog, Cat, Eagle* i *Sparrow* ujedno pripadaju i najopćenitijoj klasi *Animal*.



Slika 7.11: UML prikaz hijerarhije klasa

Naposljetku, sve ranije prikazane koncepte možemo povezati u složeni primjer. Na slici 7.12 opisan je dio informacijskog sustava nekog fakulteta. Fakultet je opisan klasom *Faculty*, koja ima atribute identifikator (*ID*), ime (*Name*) i opis (*Description*). Fakultet može sadržavati više zavoda (*Department*), pri čemu se koristi relacija kompozicije kao čvrste agregacije jer zavod ne može postojati neovisno o fakultetu. Na fakultetu se također nalaze osobe u različitim ulogama. Da bi se naglasilo da sve pripadaju fakultetu, korištena je relacija agregacije koja naglašava da svaka osoba pripada fakultetu, ali da može postojati neovisno i ako prekine vezu s fakultetom (npr. student se ispiše, zaposlenik prekine radni odnos i sl.). Klasa *Person* predstavlja općeniti predložak za osobu, koji sadrži atribute matični broj (*ID*), ime (*Name*), adresa (*Address*) i telefon (*Phone*). Iz bazne klase *Person* izvode se dvije osnovne kategorije osoba koje se nalaze u sustavu, a to su studenti (*Student*) i zaposlenici (*Staff*). Te klase specijaliziraju se tako da sadrže dodatne informacije o pojedinim kategorijama osoba pa npr. studenti imaju atribute studentski status (*Status*) i odabrani studijski program (*StudyProgramme*), a zaposlenici radni status (*Status*) i plaću (*Salary*). Među zaposlencima razlikuju se profesori (*Academic*) i pomoćno osoblje (*Admin*). Profesori imaju dodatni atribut akademskog zvanja (*AcademicTitle*). U prikazanom modelu svaki zaposlenik je povezan s uredom (klasa *Office*), pri čemu multiplicitet poveznice za klasu

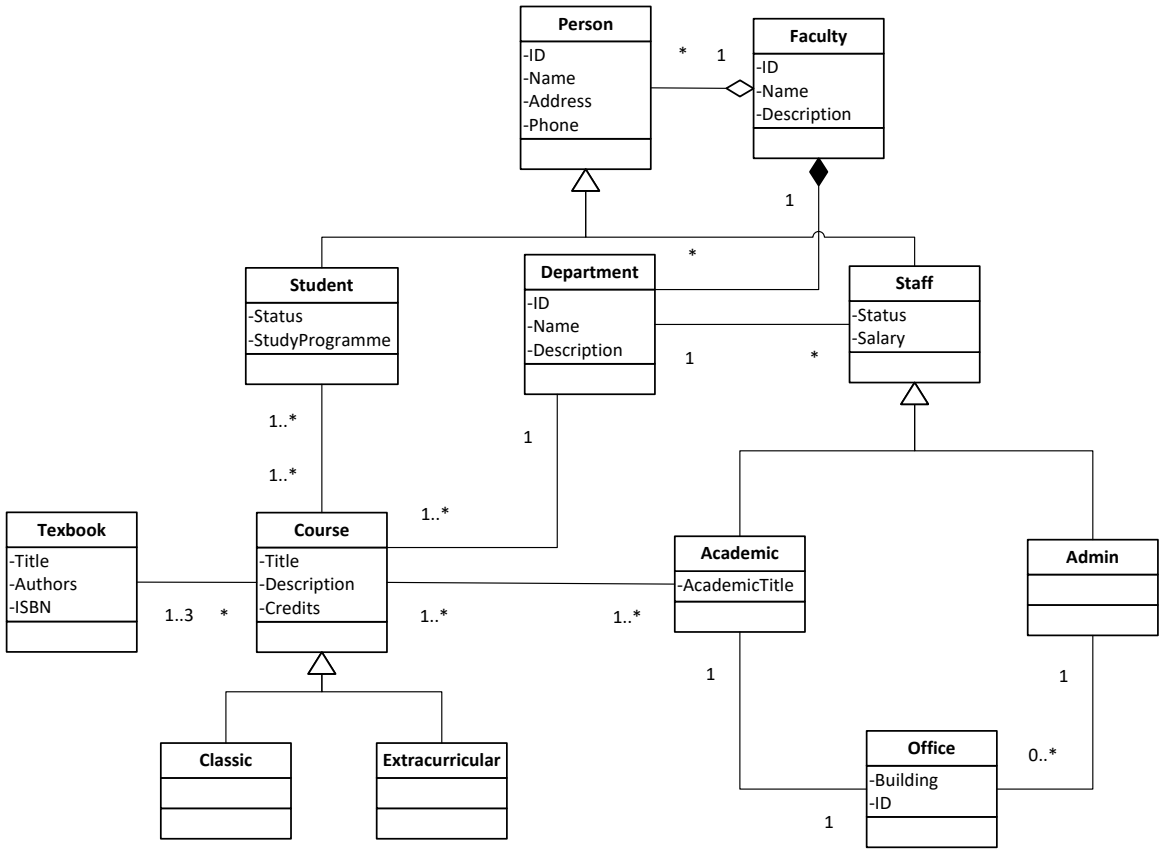
Academic upućuje da svaki profesor ima svoj ured, dok pomoćno osoblje može biti bez dodijeljenog ureda ili biti zaduženo za više prostorija. Svaki zavod nudi minimalno jedan predmet (*Course*). Svaki predmet ima atribute ime (*Title*), opis (*Description*) i broj ECTS bodova (*Credits*). Svakom predmetu pridjeljena su jedan do tri udžbenika (*TextBook*), a postoji razlikovanje između običnih predmeta (*Classic*) i izvannastavnih aktivnosti (*Extracurricular*). Zanimljivo je promotriti relacije između klasa *Student-Course* i *Academic-Course*. U oba slučaja na svakoj strani relacije nalazi se više entiteta (minimalno jedan), tako da npr. jedan student može odabrati više predmeta, a jedan predmet može biti odabran od strane više studenata. U slučaju profesora jedan profesor može predavati više predmeta, a na jednom predmetu može predavati više profesora. U oba slučaja radi se o kompleksnim relacijama između objekata koje bi bilo prikladno pobliže opisati pridruženim razredom (engl. *association class*). U prikazanom dijagramu naznačeno je da među studentima i predmetima te profesorima i predmetima postoji složena veza, ali bi u daljnjoj razradi modela bilo prikladno uvesti dva nova pridružena razreda koji bi pobliže opisali odnose između studenata i predmeta (način na koji studenti odabiru predmete) te između profesora i predmeta (način na koji se predmeti dodijeljuju pojedinim profesorima koji će ih predavati).

7.3 Dijagram objekata (engl. o*bject diagram*)

Dijagram objekata (engl. *object diagram*) izvodi se iz dijagrama razreda i služi za prikaz odnosa među *instancama* razreda u nekom promatranom trenutku izvođenja programa. Koristi se za verifikaciju ispravnosti modela prikazanog dijagramom razreda kroz proučavanje odnosa između instanci klasa u različitim testnim slučajevima. Temeljna razlika prema dijagramu razreda je u tome što dijagram razreda opisuje odnose među klasama na konceptualnoj razini, a dijagram objekata statički prikaz konkretnog stanja instanci razreda u nekom trenutku radi verifikacije modela u promatranoj situaciji.

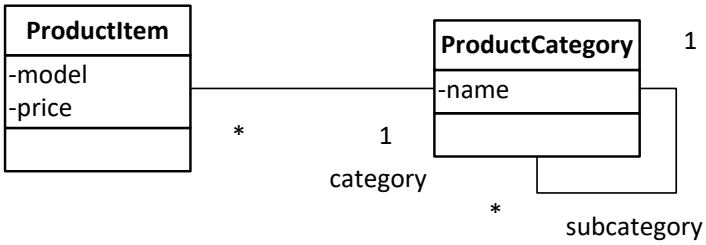
Dijagram objekata bit će ilustriran na primjeru praćenja stanja skladišta u trgovini. Na slici 7.13 prikazan je dijagram razreda koji opisuje klase za praćenje stanja skladišta u trgovini. Definirani su razredi *ProductCategory* i *ProductItem*. Klasa *ProductCategory* predstavlja kategoriju proizvoda, koja može biti hijerarhijski organizirana, što je naznačeno refleksivnim pridruživanjem na slici 7.13. Svaki tip proizvoda opisan je klasom *ProductItem*, gdje svaki

proizvod pripada određenoj kategoriji. Prikaz na slici 7.13 opisuje apstrakne veze između klasa *ProductCategory* i *ProductItem*.

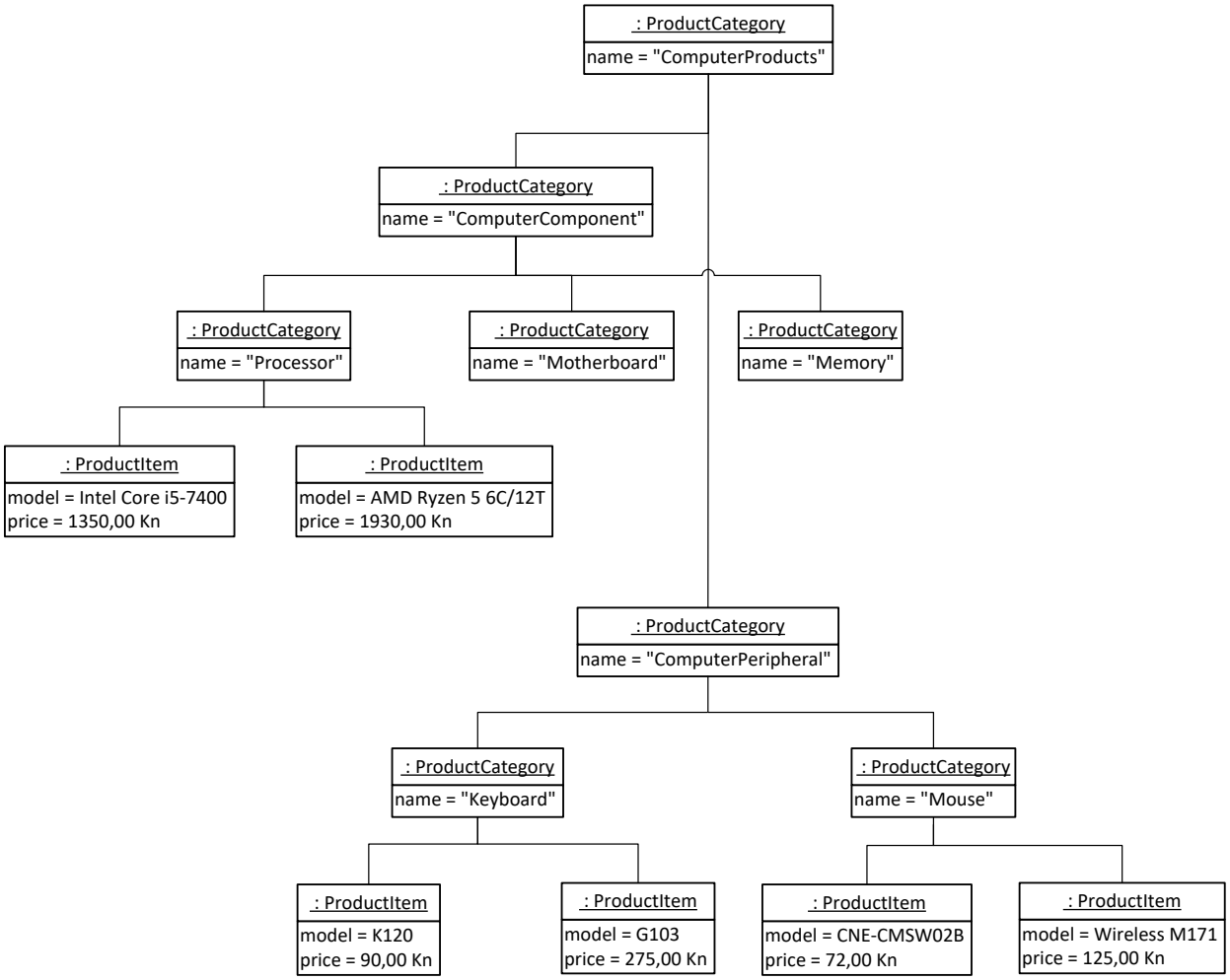


Slika 7.12: Primjer složenog UML dijagrama razreda (informacijski sustav fakulteta)

Na slici 7.14 prikazan je dijagram objekata izveden iz dijagrama razreda sa slike 7.13. Slika 7.14 prikazuje stanje skladišta u nekom trenutku. Program će prilikom definiranja nove kategorije stvoriti novu instancu klase *ProductCategory*, pri čemu je potrebno instance kategorija proizvoda organizirati hijerarhijski. U prikazanom primjeri vidi se da se radi o trgovini računalne opreme, kod koje se u ishodištu hijerarhije kategorija nalazi objekt koji označava općenito računalne proizvode („*ComputerProducts*“), a ishodišna kategorija dalje se grana u podkategorije.



Slika 7.13: UML dijagram razreda koji opisuje kategorije proizvoda i proizvode na skladištu u trgovini



Slika 7.14: UML dijagram objekata koji opisuju stanje skladišta u trgovini u nekom trenutku

Unutar kategorije „*ComputerProducts*“ nalaze se podkategorije „*ComputerComponents*“ i „*ComputerPeripherals*“, koje se kroz nove instance klase *ProductCategory* dijele na još specifičnije podkategorije (*„Processor“, „Memory“, „Motherboard“, „Keyboard“ i „Mouse“*). Na dnu hijerarhije nalaze se u krajnjim kategorijama instance klase *ProductItem*, koje predstavljaju konkretne proizvode. Primjerice, u kategoriji procesora nalaze se dva proizvoda koja su dostupna u trgovini.

Dijagrami objekata korisni su u analizi dijagrama razreda promatranjem specifičnih testnih slučajeva koje je potrebno pažljivo odabrati tako da reprezentiraju određene situacije interesantne za analizu s obzirom da se iz jednog dijagrama razreda može izvesti beskonačno dijagrama objekata. Može se uočiti sličnost u notaciji koja se koristi kod dijagrama razreda i objekata, s tom razlikom da se u slučaju dijagrama objekata kod objekata u zaglavlju navodi ime razreda (podcrtano u prikazu) i trenutne vrijednosti atributa instance. Također se za razliku od dijagrama klasa ne koriste relacije višestrukosti, već samo jednostavne veze među objektima, jer je u dijagramu objekata potrebno prikazati sve konkretne instancirane objekte i veze među njima.

7.4 Dijagram obrazaca uporabe (engl. *use case diagram*)

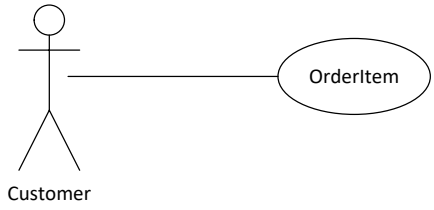
Do sada prikazani dijagrami klasa i objekata služili su strukturno modeliranje, odnosno tzv. *statički pogled* u sustav koji prikazuje građu i unutarnju organizaciju gradivnih jedinica sustava. Dijagram obrazaca uporabe predstavlja drugačiji pogled na sustav koji služi da modeliranje ponašanja sustava, odnosno predstavlja jedan od tzv. *dinamičkih pogleda* u sustav.

Obrasci ili slučajevi uporabe (engl. *use cases*) predstavljaju niz aktivnosti ili interakcija između korisnika i sustava koje je potrebno obaviti da bi se ostvario određeni cilj. Obrasci uporabe koriste se u UML modeliranju za opis funkcionalnosti i zahtjeva na sustav iz perspektive koju vidi korisnik sustava. Opisuje se *što* je potrebno postići u smislu funkcionalnosti sustava, ali ne i *kako* tu funkcionalnost ostvariti na implementacijskoj razini.

UML dijagrami obrazaca uporabe sastoje se od tri najvažnija elementa:

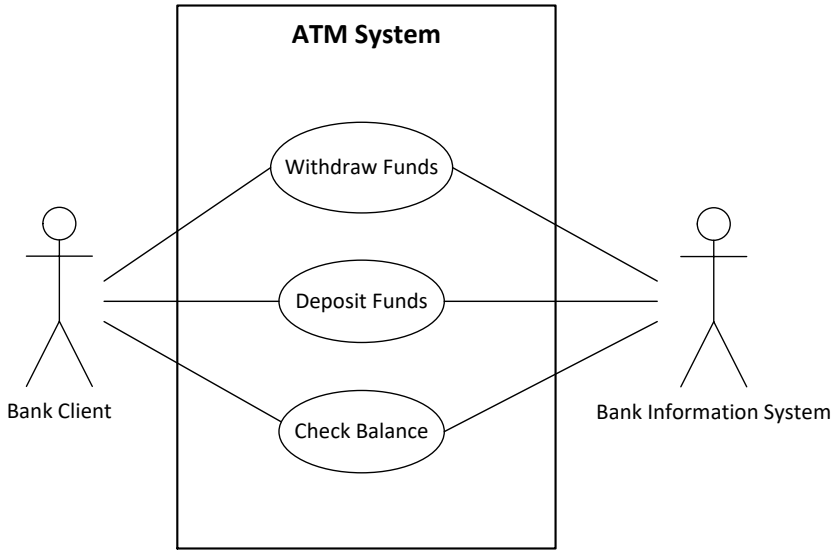
- obrazac uporabe (engl. *use case*) – specifična funkcionalnost koju sustav treba omogućiti,
- korisnik sustava ili agent (engl. *actor*) – entitet koji upravlja akcijama sustava, koji u tipičnom slučaju predstavlja osobu - korisnika sustava, ali u općem slučaju to mogu biti i drugi računalni sustavi koji imaju sposobnost upravljanja promatranim sustavom,
- elementi implementacije obrasca uporabe (engl. *subjects*).

Obrasci uporabe se u UML dijagramu predstavljaju elipsom u kojoj se obavezno navodi ime obrasca uporabe. Korisnici sustava predstavljaju se stiliziranom figurom čovjeka, a u općem slučaju se može raditi i o drugom računalnom sustavu, bez obzira na reprezentaciju simbola. Primjer jednostavnog UML dijagrama obrasca uporabe u kojem kupac obavlja narudžbu prikazan je na slici 7.15. Na slici korisnik tipa *Customer* obavlja postupak narudžbe predstavljen obrascem uporabe *OrderItem*. Između korisnika (agenta) i obrasca uporabe postoje jednostavna veza prikazana ravnom linijom.



Slika 7.15: Primjer jednostavnog dijagrama obrasca uporabe

Obrasci uporabe predstavljaju funkcionalnost određenog subjekta unutar sustava. Sve što nije dio subjekta čija se funkcionalnost opisuje potrebno je modelirati agentom (*actor*). Na taj način može se dobro odijeliti što je opseg odgovornosti sustava ili podsustava, a o čemu brinu drugi sustavi ili korisnici (agenti). Kako bi se označile granice sustava u UML dijagramu obrazaca uporabe koristi se pravokutnik, u kojem se nalazi naziv modeliranog sustava (ili podsustava) na kojeg se obrasci uporabe odnose. Ove smjernice ilustrirane su u primjeru na slici 7.16.



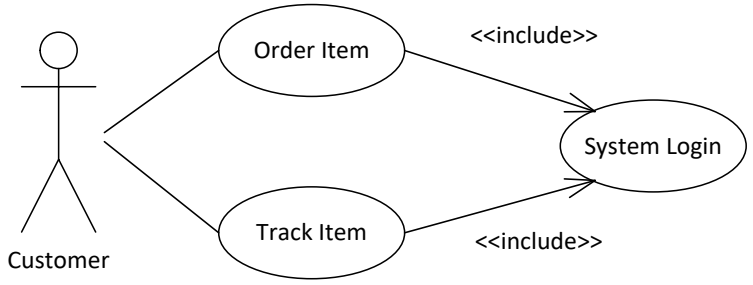
Slika 7.16: Dijagram obrazaca uporabe bankomata

Slika 7.16 prikazuje dijagram obrazaca uporabe sustava bankomata, gdje su naznačene granice sustava koji se opisuje (*ATM System*), definiraju se dva agenta (*actors*) *Bank Client* (osoba koja koristi bankomat) i *Bank Information System* (računalni sustav banke) te tri obrasca uporabe (podizanje novca (*Withdraw Funds*), uplata novca (*Deposit Funds*) i provjera stanja (*Check Balance*)). Iz vizualnog prikaza odmah je jasno gdje su granice sustava i koji su obrasci uporabe sustava te koji agenti sudjeluju u ostvarenju funkcionalnosti sustava.

Opisane mogućnosti prikaza obrazaca uporabe UML dodatno proširuje kako bi se efikasnije mogli opisati složeniji scenariji funkcionalnosti. Nadogradnja opisa obrazaca uporabe moguća je kroz *uključivanje* (<<*include*>>) ili *proširenje* (<<*extend*>>) obrasca uporabe, a obrasce uporabe i agente također je moguće *generalizirati*.

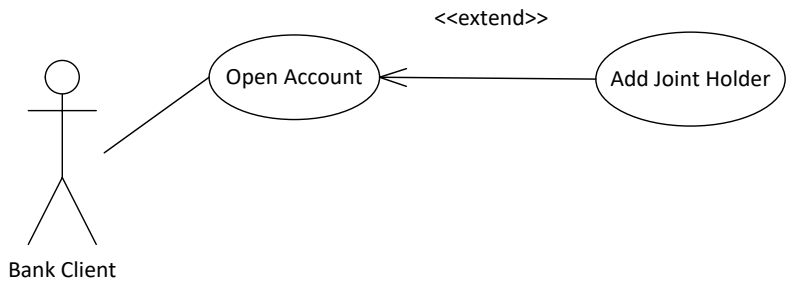
Uključivanje obrasca uporabe (engl. *use case inclusion*) predstavlja situaciju kada je *nužno* u novi obrazac uporabe uključiti neki postojeći da bi se operacija mogla izvesti. Primjer je prikazan na slici 7.17 gdje su prikazana dva obrasca uporabe, naručivanje proizvoda (*Order Item*) i praćenje proizvoda (*Track Item*), pri čemu se niti jedan od njih ne može u cjelini izvršiti ako ne uključi izvođenje posebno definiranog obrasca uporabe prijave na sustav (*System Login*). Ovakvo razdvajanje obrazaca uporabe omogućuje lakše dijeljenje nekih

obrazaca uporabe (u ovom slučaju prijave u sustav), a naglašava se nužnost uključenja dijeljenog obrasca uporabe kako bi se funkcionalnost mogla izvršiti u cjelini.



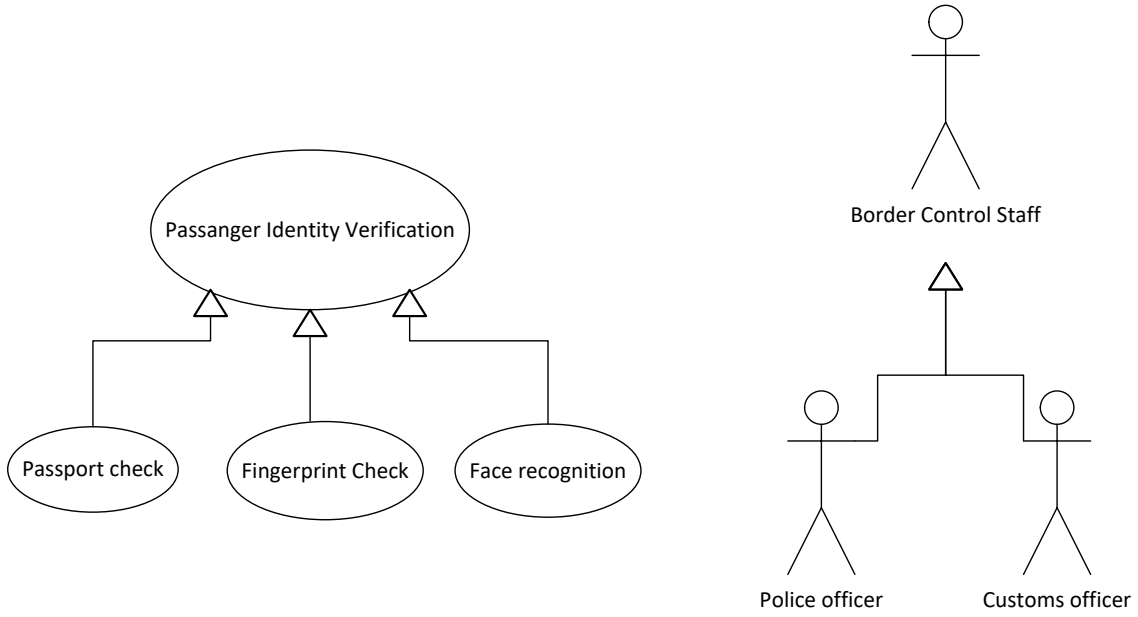
Slika 7.17: Dijagram obrasca uporabe – *use case inclusion*

Obrazac uporabe može se proširiti (engl. *use case extension*) u slučaju kada promatrani obrazac uporabe može samostalno funkcionirati bez pomoći drugog obrasca uporabe, osim u nekim posebnim slučajevima. Primjer proširenja obrasca uporabe prikazan je na slici 7.18 gdje je prikazan obrazac uporabe otvaranja računa od strane novog korisnika u banci. Korisnik (*Bank Client*) može otvoriti račun i za tu operaciju (*Open Account*) nije mu potrebna pomoć od drugog obrasca uporabe. Međutim, u posebnom slučaju kada korisnik želi dodati dodatnog korisnika na svoj račun prilikom otvaranja novog računa, osnovni obrazac uporabe otvaranja novog korisnika (*Open Account*) ne može riješiti tu situaciju bez pomoći posebnog obrasca uporabe dodavanja dodatnog korisnika (*Add Joint Holder*), što je naznačeno posebnim obrascem uporabe u dijagramu na slici 7.18. I u ovom slučaju korisno je imati razdvojene obrasce uporabe kako bi ih mogli lakše dijeliti i drugi obrasci uporabe među sobom. Osnovna razlika između <<*include*>> i <<*extend*>> slučajeva je da kod uključenja (*include*) se osnovni obrazac uporabe ne može nikada izvesti bez pomoći uključenog obraca, a kod proširenja (*extend*) se osnovni obrazac uporabe u pravilu može izvesti samostalno, osim u posebnim slučajevima kada mu je potrebna pomoć drugog obrasca uporabe. Uočiti da su smjerovi strelica na slikama 7.17 i 7.18 suprotni.



Slika 7.18: Dijagram obrasca uporabe – *use case extension*

Generalizacija se koristi na sličan način kao kod dijagrama razreda tako da se iz općenitijih obrazaca uporabe (*use cases*) izvode specifičniji, ili da se iz općenitije definiranih agenata (*actors*) izvode specifičniji. Na slici 7.19 prikazan je primjer poopćenja obrazaca uporabe koji se odnose na način kontrole identiteta putnika na graničnom prijelazu te primjer poopćenja vrste agenata koji mogu sudjelovati kao osoblje u provjeri identiteta putnika.



Slika 7.19: Dijagram obrasca uporabe – *use case generalization*

Dijagrami obrazaca uporabe izuzetno su važni u fazi prikupljanja specifikacija jer omogućuju učinkovit opis sustava iz pogleda korisnika sustava. Služe kao vrijedan alat za identifikaciju svih dionika koji će koristiti sustav (*actors*) te načina na koji će se sustav koristiti i koje funkcionalnosti treba pružiti (*use cases*). Dijagrami obrazaca uporabe opisuju događaje u sustavu, ali ne bave se opisom kako implementirati funkcionalnost. Predstavljaju

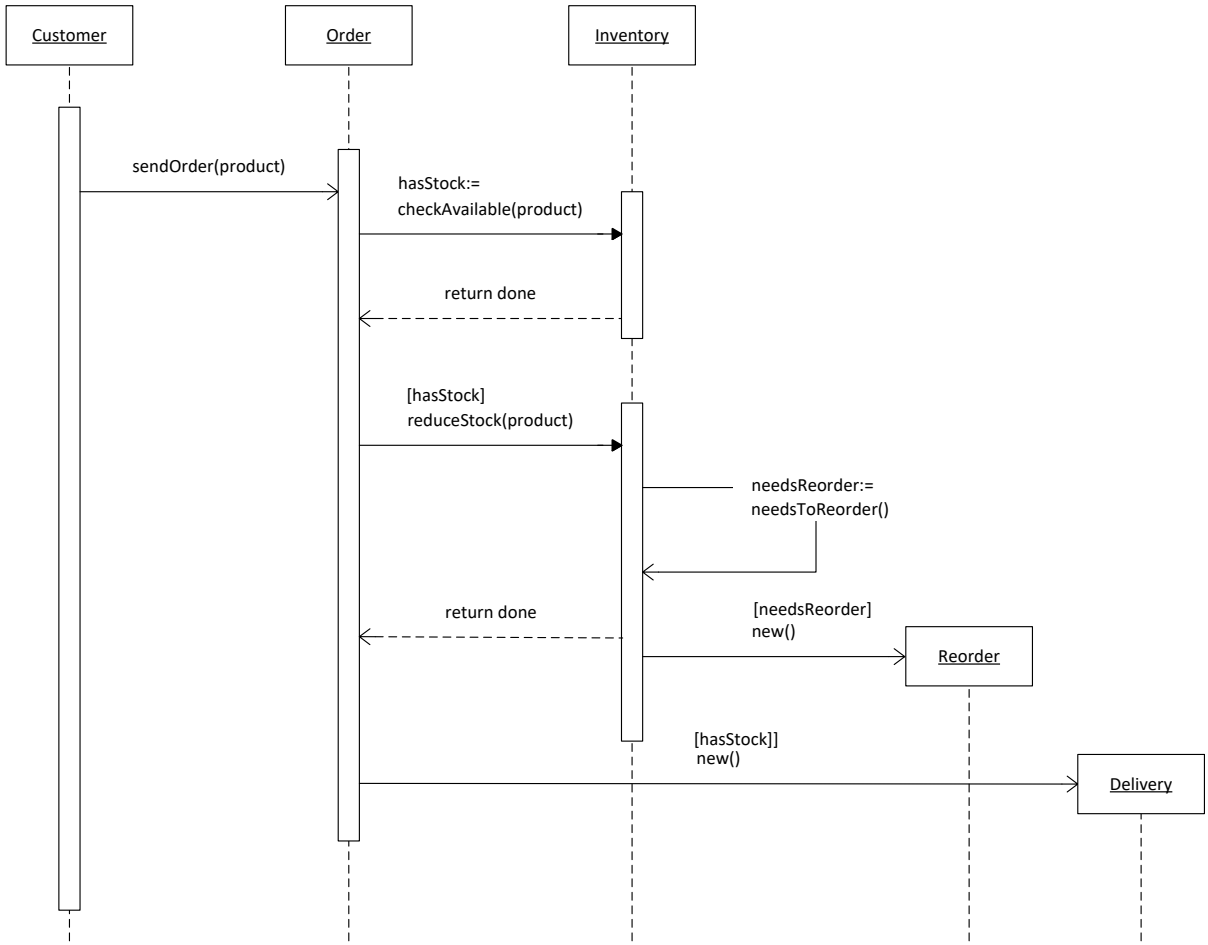
opis ponašanja i dinamičkog pogleda na sustav na visokoj razini apstrakcije, a dublji uvid kako implementirati funkcionalnost i izgraditi arhitekturu rješenja potrebno je razraditi drugim vrstama UML dijagrama.

7.5 Interakcijski dijagrami (engl. *interaction diagrams*)

Interakcijski dijagrami (engl. *interaction diagrams*) modeliraju međusobnu surađuju između objekata u sustavu s ciljem ostvarenja neke funkcionalnosti. Predstavljaju drugi vid dinamičkog pogleda u sustav kao i ranije opisani dijagrami obrazaca uporabe, a tipično se koriste za detaljniji opis ponašanja jednog odabranog obrasca uporabe. Interakcijski dijagrami temelje se na prikazu objekata koji sudjeluju u ostvarenju određene funkcionalnosti sustava i poruka koje se šalju između njih. Dok dijagrami obrazaca uporabe govore o tome *što* treba sustav raditi u interakciji s korisnikom, interakcijski dijagrami gore *kako* tu funkcionalnost ostvariti međusobnom interakcijom i suradnjom među objektima. Postoje dvije vrste UML interakcijskih dijagrama, a to su sekvencijski dijagrami (engl. *sequence diagram*) i kolaboracijski dijagrami (engl. *collaboration diagram*). Svaki od te dvije vrste dijagrama modelira ponašanje objekata na sličan način, ali uz nešto drugačiji prikaz i notaciju.

Primjer sekvencijskog dijagrama prikazan je na slici 7.20. Dijagram opisuje postupak narudžbe proizvoda. Na dijagramu se mogu vidjeti objekti *Customer*, *Order*, *Inventory*, *Reorder* i *Delivery*. Objekti su naznačeni pravokutnicima u kojima se nalazi naziv klase, a moguće je dodati i naziv instance objekta. Objekti ostvaraju funkcionalnost tako da razmjenjuju *poruke*, koje zapravo predstavljaju funkcijske pozive između objekata. Primjerice, objekt *Customer* započinje narudžbu slanjem poruke *sendOrder(product)* prema objektu *Order*. Okomite iscrtkane linije prikazuju crtu vremena života objekta (*lifeline*), a svi odnosi i protokol razmjene poruka promatra se u vremenu gdje se kasniji trenuci nalaze u donjem dijelu dijagrama. Ako se eksplicitno želi naglasiti vremenski period kada objekt postoji i kada je aktivan, to se naznačuje uskim pravokutnikom preko *lifeline*-a (tzv. *activation bar*). Na dijagramu je naznačen strelicom smjer slanja poruka, a položajem na *lifeline*-u kronološki raspored slanja poruka. Poruke imaju imena koja tipično odgovaraju

imenima funkcija, a mogu sadržavati parametre. Također je ponekad korisno pratiti rezultat izvođenja poruke varijablom, primjerice prilikom provjere stanja skladišta porukom *checkAvailable*(*product*) pamti se rezultat u varijabli *hasStock* koja upućuje na stanje skladišta ima li traženog proizvoda ili ne. Vrijednosti varijabli mogu se koristiti za *uvjete* koji moraju biti ispunjeni da bi se neka poruka poslala, što se naznačuje u uglatim zagradama. Npr. poruka *reduceStock*(*product*) da se naručeni proizvod povuča sa skladišta bit će pozvana samo ako je zadovoljen uvjet *[hasStock]*. Crtkanim linijama naznačen je povratak izvornom objektu nakon slanja poruke (*return*), što nije nužno radi preglednosti naznačavati za svaku poruku, već samo tamo gdje se procijeni da je radi bolje preglednosti poželjno. Postoji mogućnost i da objekt pošalje poruku sam sebi (*self call*) npr. nakon povlačenja jednog ili više proizvoda iz skladišta potrebno je automatski provjeriti je li potrebno naručiti od dobavljača nove proizvode kako se zalihe na skladištu ne bi ispraznile.



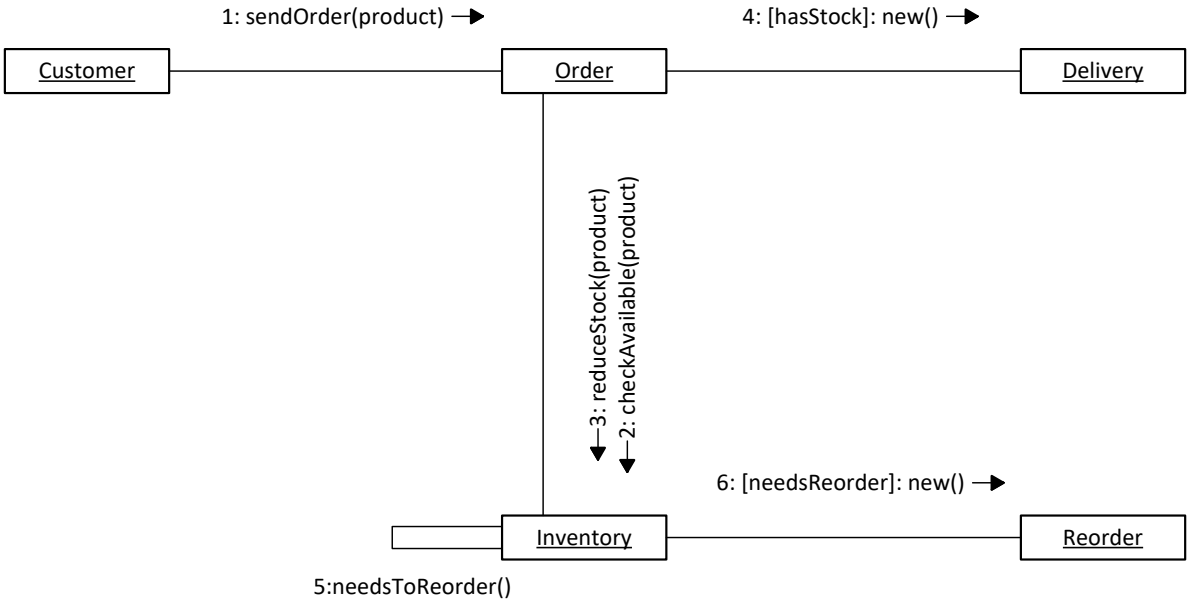
Slika 7.20: Sekvencijski dijagram postupka narudžbe proizvoda

Prilikom slanja poruka kada je važno naglasiti da objekt mora čekati na odgovor prije nego što nastavi dalje s bilo kakvim poslom koristi se strelica s ispunjenim vrhom (npr. kod poruke *checkAvailable*). To je tzv. sinkroni (*synchronous*) ili blokirajući poziv (tipično realiziran kao poziv funkcije), a kada to nije važno (npr. *Customer* može raditi i nešto drugo za vrijeme dok traje narudžba) koriste se tzv. asinkroni (*asynchronous*) ili neblokirajući pozivi putem poruka, što se naznačuje drugim tipom strelice na poruci (otvorena bez ispune).

Na temelju navedenih obrazloženja simbola i značenja elemenata u sekvencijskom dijagramu može se interpretirati na koji je način implementirana funkcionalnost obrasca uporabe (*use case*) prikazanog na slici 7.20. Obrazac uporabe odnosi se na postupak narudžbe proizvoda,

pri čemu korisnik (*Customer*) naručuje proizvod (*Order*, poruka *sendOrder()*), nakon čega se najprije provjerava stanje skladišta (*Inventory*, poruka *checkAvailable()*) i ako je proizvod (*product*) na skladištu dostupan (uvjet *hasStock*), proizvod se najprije povlači sa skladišta (poruka *reduceStock*), a onda se stvara novi objekt za isporuku korisniku (*Delivery*), slanjem poruke *new()* koja će taj objekt i stvoriti (vidi se na dijagramu da *lifeline* objekta *Delivery* započinje pri dnu). Dodatno, objekt *Inventory* automatski provjerava stanje skladišta (poruka *needsToReorder*) i ako je potrebno (ispunjen uvjet *needsReorder*) automatski kreira objekt koji će se pobrinuti za nadopunu zaliha na skladištu (slanje poruke *new()* za kreiranje objekta *Reorder*).

Drugi način opisa implementacije funkcionalnosti obrasca uporabe narudžbe proizvoda prikazanog na slici 7.20 je korištenjem *kolaboracijskog* dijagrama, koji je prikazan na slici 7.21. Na slici 7.21 mogu se uočiti isti objekti i poruke kao i na slici 7.20, ali su interakcije prikazane na drugačiji način. Dijagram kolaboracije prikazuje objekte i poruke koje međusobno razmjenjuju, ali naglasak nije na vremenskoj uređenosti prikaza protokola razmjene poruka, već na strukturi interakcija iz koje je vidljivo koji objekt kome šalje poruke. Umjesto fokusa na vremenski raspored nastoji se što jasnije prikazati međudjelovanje pojedinih objekata. U dijagramu kolaboracije prikazan je smjer razmjene poruka, nazivi poruka i uvjeti kao i u sekvencijskom dijagramu, a također se koriste redni brojevi ispred naziva poruke da se prikaže uređeni redoslijed slanja poruka. Iz sekvencijskog dijagrama lakše je vidjeti kada se poruke šalju, a iz kolaboracijskog koji objekti međusobno razmjenjuju poruke.



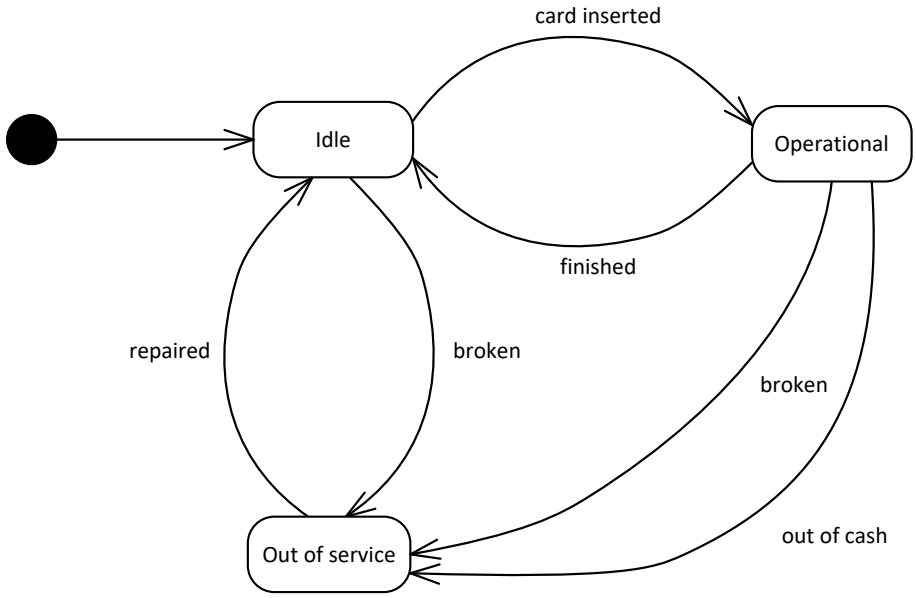
Slika 7.21: Kolaboracijski dijagram postupka narudžbe proizvoda

7.6 Dijagram stanja (engl. *statechart diagram*)

Dijagram stanja (engl. *statechart diagram*) služi za dinamičko modeliranje ponašanja sustava kroz pogled na moguća stanja objekata u sustavu i načine na koji objekti prelaze između stanja. Dok se sekvencijski i kolaboracijski dijagrami fokusiraju na postupke i korake koje objekti trebaju izvršiti da bi se ostvarila određena funkcionalnost definirana obrascem uporabe, dijagram stanja modelira posljedice ponašanja objekata koji sustav ili podsustave dovode u određena stanja kroz radnje koje obavljaju ili kao posljedicu odgovora na podržaje izvan sustava.

UML dijagram stanja usko je povezan s pojmom automata s konačnim brojem stanja (engl. *finite state machine*, FSM). Automat s konačnim brojem stanja čini skup diskretnih *stanja* u kojima se može nalaziti sustav te *događaji* koji okidaju *prijelaze* među stanjima. Na slici 7.22 prikazan je jednostavan dijagram stanja koji prikazuje funkcioniranje bankomata. Svako stanje označava se u dijagramu pravokutnikom sa zaobljenim rubovima u kojeg se upisuje naziv stanja. Strelice označavaju pravac prijelaza između stanja, a na njima je naveden događaj (uvjet) koji uzrokuje prijelaz između stanja. Posebno stanje u dijagramu predstavlja

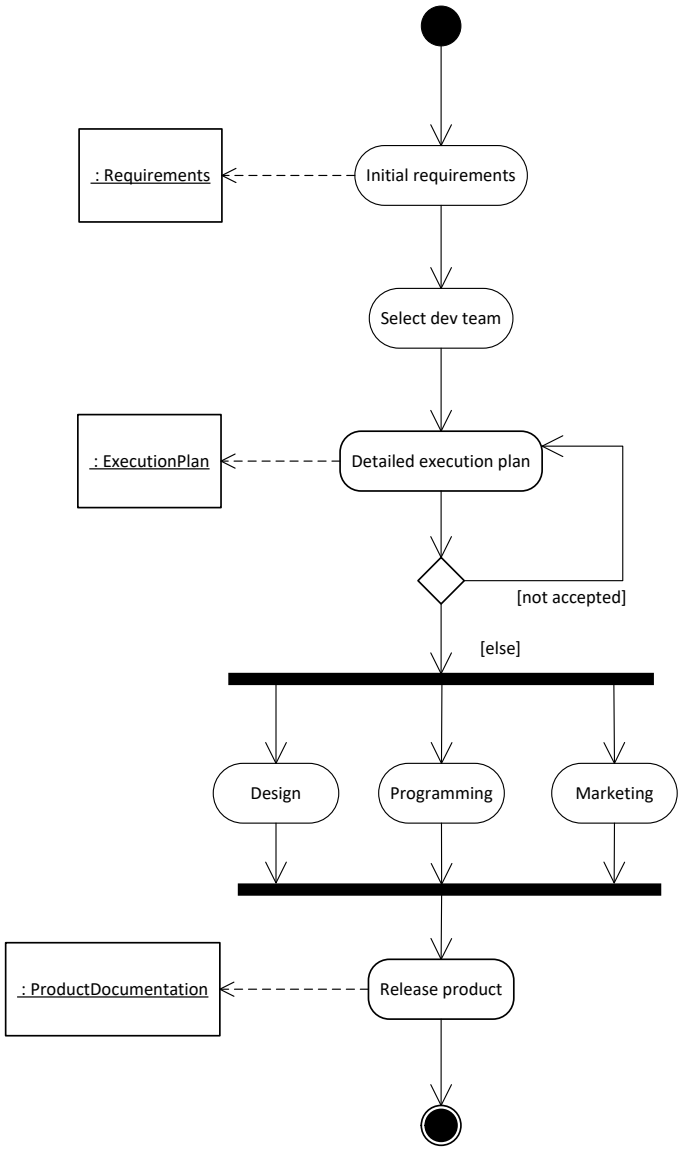
početno stanje, koje je označeno simbolom ispunjenog kruga. U prikazanom primjeru bankomat inicijalno ulazi u stanje pripravnosti (*Idle*) i čeka da korisnik umetne karticu (događaj *card inserted*). Kada se to dogodi, prelazi u stanje *Operational* u kojem korisnik od bankomata traži određene funkcije (npr. ispis stanja računa, podizanje gotovine ili uplatu gotovine). Kada korisnik prekine s radom (događaj *finished*) bankomat se ponovo vraća u stanje *Idle* i čeka da novi korisnik umetne karticu. U bilo kojem od stanja *Idle* ili *Operational* može doći do kvara (događaj *broken*), pri čemu bankomat prelazi u stanje u kojemu je izvan upotrebe (*Out of service*). Iz tog stanja može izaći samo ako se poduzme odgovarajuća akcija (označena kao prijelaz *repaired*), nakon čega je bankomat opet u funkciji i u stanju *Idle*. Moguć je prijelaz iz stanja *Operational* u stanje *Out of service* ako bankomat nakon podizanja gotovine ostane bez novca za isplatu, što je također potrebno riješiti kako bi ga se vratilo natrag u stanje *Idle* (označeno kao događaj *repaired*).



Slika 7.22: Jednostavni dijagram stanja bankomata

Dijagrami stanja pogodni su za modeliranje ponašanja sustava upravljanog događajima (*event-driven behaviour*) i modeliranje protokola (npr. komunikacijskih, poslovnih procedura i sl.).

7.7 Dijagram aktivnosti (engl. *activity diagram*)



Slika 7.23: Dijagram aktivnosti razvoja programskog proizvoda

Dijagram aktivnosti (engl. *activity diagram*) služi za dinamičko modeliranje ponašanja sustava i fokusira se na razmjenu poruka između *aktivnosti*. Primjenjuje se za modeliranje tijeka upravljanja (engl. *control-flow transitions*) i tijeka podataka (engl. *object-flow transitions*). UML dijagrami aktivnosti slični su dijagramima tijeka (engl. *flowchart*) koji su se tradicionalno prije uvođenja UML metodologije koristili za opisivanje algoritama, ne samo u domeni razvoja programske potpore, već i u drugim područjima (tipično u procesu

modeliranja općenitih poslovnih procesa). UML dijagrami aktivnosti predstavljaju nadogradnju ranije poznatih koncepata iz dijagrama tijeka, s time da uvode dodatne mogućnosti opisane UML standardom.

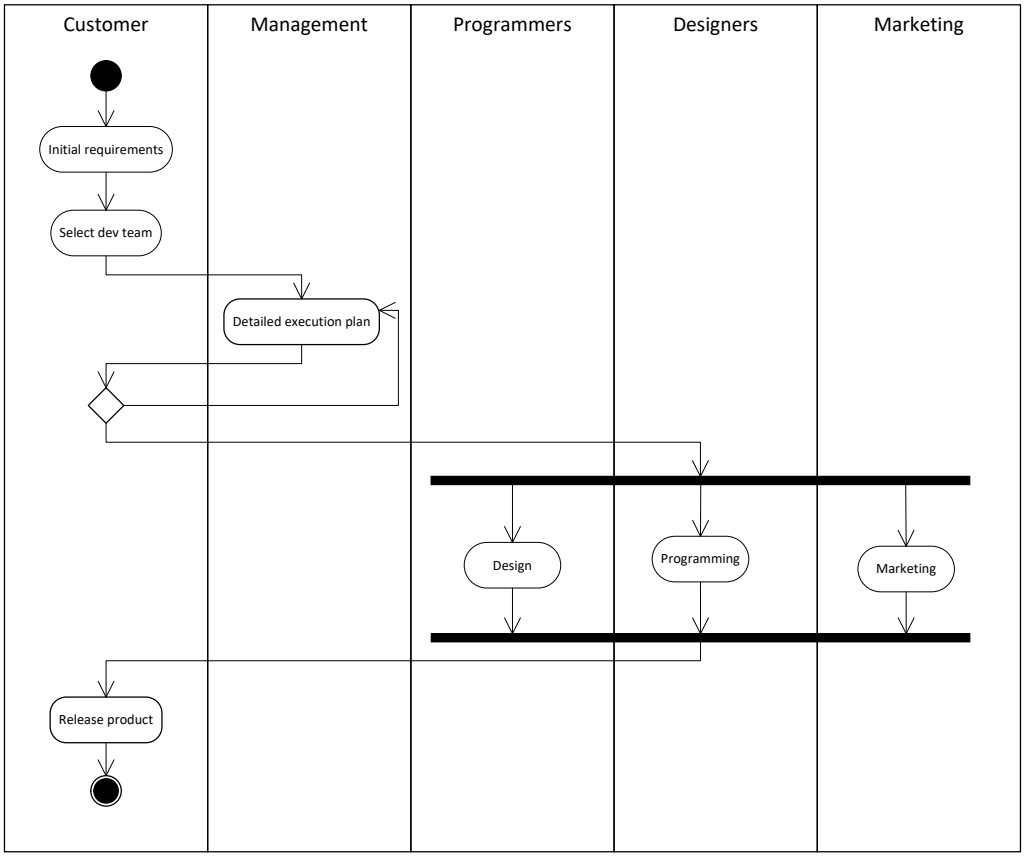
Dijagram aktivnosti bit će opisan na primjeru razvoja programskog proizvoda kako je prikazano na slikama 7.23 i 7.24. Na slici 7.23 aktivnosti su opisane pravokutnicima sa zaobljenim rubovima u kojima pišu nazivi aktivnosti i među kojima postoje poveznice (*control-flow*), dok su običnim pravokutnicima označeni objekti koji sadrže podatke vezane uz pojedine aktivnosti (*object-flow*). Smjer strelica između aktivnosti upućuje na redoslijed njihovog izvođenja. Mali romb prikazuje simbol uvjetnog grananja, a debele crte označavaju mjesta gdje se jedna aktivnost grana u više novih aktivnosti koje se izvode *paralelno* u vremenu (točka račvanja, engl. *concurrent fork*) i mjesto gdje se te aktivnosti skupljaju nakon što su obavljene (engl. *concurrent join*), što je preduvjet da se može nastaviti sljedeća aktivnost. U dijagramu aktivnosti nova aktivnost započinje nakon što je završena prethodna, odnosno više njih ako se nova aktivnost u dijagramu nalazi iza točke skupljanja paralelnih aktivnosti. Posebnim simbolima (ispunjenim krugovima) označeni su početno i završno stanje u dijagramu aktivnosti.

Postupak razvoja programskog proizvoda prikazan dijagramom aktivnosti na slici 7.23 može se protumačiti na sljedeći način. Najprije naručitelj projekta treba kroz aktivnost *Initial requirements* definirati kakav proizvod želi razviti, što dokumentira u dokumentu *Requirements* prikazanom na dijagramu. Na temelju poznatih okvirnih parametara proizvoda prelazi na aktivnost pronalaženja i odabira razvojnog tima (aktivnost *Select dev team*), jer se pretpostavlja da mu je potrebna vanjska usluga. Razvojni tim na temelju svojeg ekspertnog znanja izrađuje detaljni provedbeni plan projekta (*ExecutionPlan*), ako je potrebno i u više iteracija dok plan ne bude prihvatljiv korisniku (što je naznačeno uvjetnim grananjem). Nakon što korisnik prihvati detaljni provedbeni plan projekta, započinje razvoj na proizvodu gdje se paralelno obavljaju tri aktivnosti:

- dizajn proizvoda (*Design*) – definiranje korisničkog sučelja, izrada *wireframe* modela, rasporeda elemenata u pogledima korisničkog sučelja, optimiranje ergonomije sučelja, izrađuju se crteži, modeli koji se prikazuju u programu i sl.,

- programiranje (*Programming*) – programiranje temeljne funkcionalnosti produkta, interakcije s korisnikom putem korisničkog sučelja, pohrane i dohvata podataka iz baze, povezivanja s vanjskim programskim servisima itd.,
- marketing (*Marketing*) – priprema korisničkih uputa, promotivnih letaka, materijala za medije, web stranice, promidžbenih filmova, reklama i sl.

Na dijagramu aktivnosti naznačeno je da se sve tri aktivnosti odvijaju paralelno, tj. i aktivnosti povezane s marketingom rade se istodobno s razvojem produkta kako bi se minimiziralo vrijeme izlaska na tržište. Nakon što je razvoj proizvoda gotov, ulazi se u aktivnost izbacivanja proizvoda na tržište, a u tom trenutku se može izraditi konačna projektna dokumentacija koja uključuje sav programski kôd, multimedijske sadržaje, dokumente i sl. koji su bitni za stavljanje proizvoda na tržište, održavanje produkta, korisničku podršku i sl.

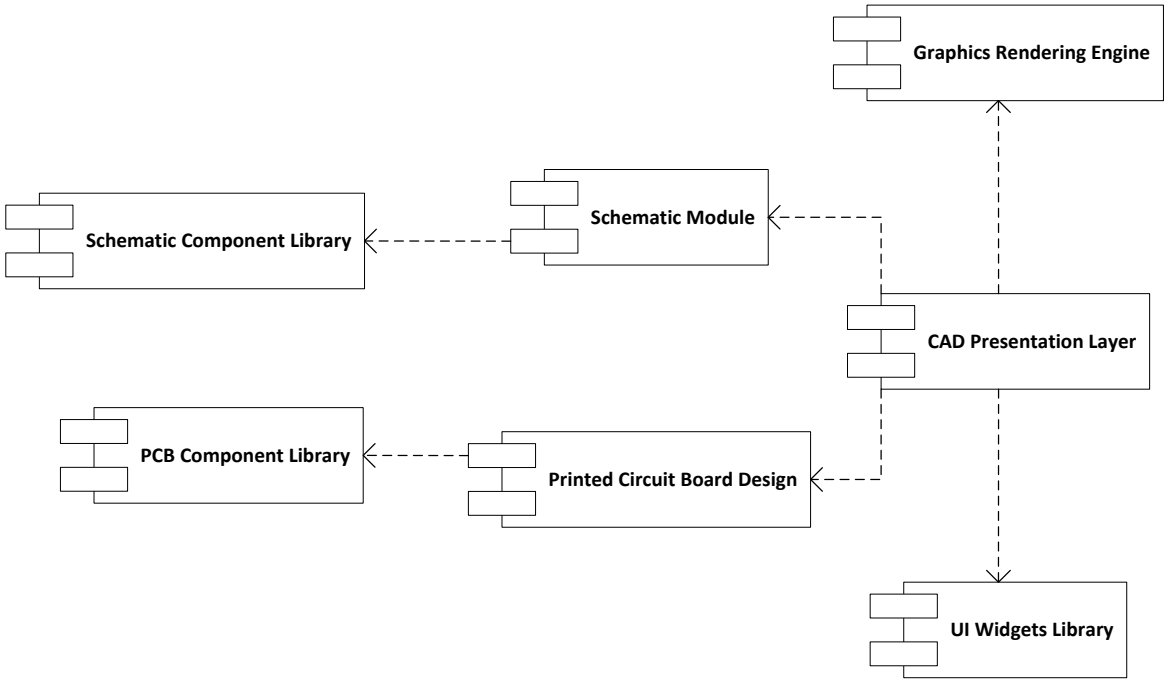


Slika 7.24: Dijagram aktivnosti razvoja programskog proizvoda (prikaz s dodijeljenim odgovornostima za aktivnosti)

Ponekad je korisno na dijagramu aktivnosti osim samog prikaza modela toka upravljanja i podataka napraviti particioniranje dijagrama da se lakše vizualiziraju aktivnosti koje čine jednu logičku cjelinu ili za koje je odgovoran neki sudionik u sustavu. Te se particije dijagrama aktivnosti nazivaju *plivačke staze* (engl. *swimlanes*), a isti dijagram aktivnosti sa slike 7.23 prikazan je na slici 7.24 na malo drugačiji način, pri čemu se iz toga prikaza jasno vidi tko je zadužen za koju od aktivnosti. Tako se može vidjeti da su uvedene uloge (*roles*) pod nazivima *Customer* (naručitelj projekta), *Management* (uprava tvrtke koja radi razvoj za naručitelja projekta), *Programmers* (programeri u razvojnoj tvrtki), *Designers* (dizajneri u razvojnoj tvrtki) i *Marketing* (marketinški stručnjaci u razvojnoj tvrtki). U prikazanom dijagramu aktivnosti *Customer* sastavlja inicijalni prijedlog projekta, odabire tvrtku koja će raditi razvoj, prihvaća prijedlog detaljnog provedbenog plana projekta i lansira produkt na tržište. *Management* kroz pregovore s *Customerom* sastavlja detaljni provedbeni plana projekta, a na razvoju proizvoda rade grupe *Programmers*, *Designers* i *Marketing*.

7.8 Dijagram komponenata (engl. c*omponent diagram*)

Dijagram komponenata (engl. *component diagram*) predstavlja statički pogled na strukturu sustava koji opisuje organizaciju i međuovisnost programskih komponenti koje implementiraju funkcionalnost programske potpore. Dijagrami komponenata opisuju arhitekturu programske potpore na visokoj razini apstrakcije. Komponente predstavljaju različite vrste modula u kojima je fizički smješten programski kôd ili druge informacije (npr. izvršne datoteke, biblioteke, konfiguracijske datoteke, baze podataka itd.).



Slika 7.25: Dijagram komponenata (CAD program)

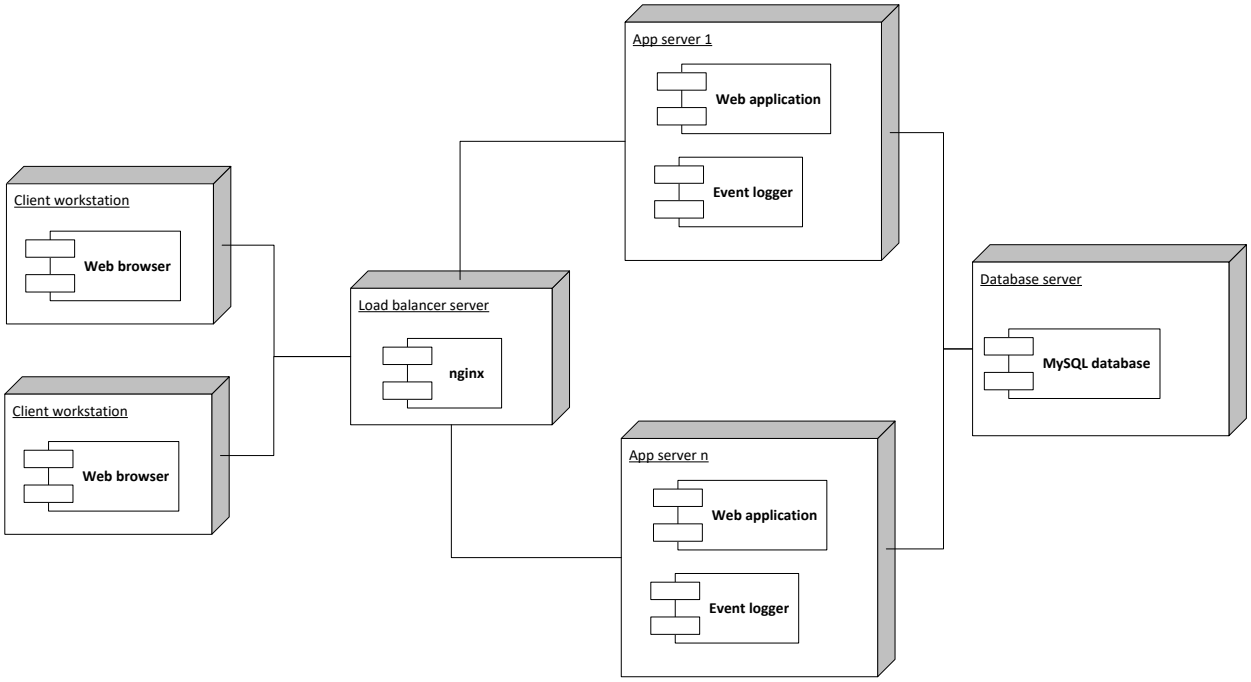
Na slici 7.25 prikazan je dijagram komponenata na primjeru CAD programa za projektiranje elektroničkih uređaja. S obzirom da su takvi programi izuzetno složeni, potrebno je identificirati od kojih će se komponenata sastojati, koje će se tehnologije uključiti prilikom razvoja programskog rješenja te kakva je međuovisnost komponenti u sustavu. Prikazana je organizacija programske potpore koja opisuje dio sučelja prema korisniku, za što je zadužena komponenta *CAD presentation Layer*. Korisnik može u programu uređivati električne sheme ili tiskane pločice putem sučelja koje nudi komponenta *CAD presentation Layer*. Za to su joj potrebne specifične funkcionalnosti koje implementiraju posebne komponente specijalizirane za uređivanje električnih shema (komponenta *Schematic Module*) i tiskanih pločica (komponenta *Printe Circuit Board Design*). Svaka od tih dviju programskih komponenti nadalje ovisi o bibliotekama u kojima su definirani simboli koji se crtaju, a koje su prikazane kao posebne komponente (*Schematic Component Library* za elektroničke komponente za električnu shemu i *PCB Component Library* za fizički prikaz komponenti na tiskanoj pločici). *CAD presentation Layer* također zahtijeva funkcionalnosti niske razine za iscrtavanje osnovnih grafičkih primitiva (npr. povezivanjem s API-jem (*Application Program Interface*) specijaliziranih biblioteka, npr. OpenGL, DirectX i sl.). Također su u

takvim programima važne neki nestandardni elementi korisničkog sučelja koji se u ovom primjeru nalaze u biblioteci *UI Widgets Library*.

7.9 Dijagram razmještaja (engl. *deployment diagram*)

Dijagram razmještaja (engl. *deployment diagram*) mapira programske komponente sustava na fizička računala na kojima će se izvoditi. Omogućuje strukturno modeliranje koje opisuje okolinu u kojoj se izvodi programska potpora i predstavlja još jednu vrstu statičkog pogleda na sustav. Za razliku od dijagrama komponenti, dijagram razmještaja prikazuje vanjske resurse koji su potrebni programskim komponentama za rad. Razmatranje dijagrama razmještaja važno je tijekom razvojnih aktivnosti kako bi se unaprijed predvidjela infrastuktura potrebna za pripremu raspoloživosti programskog sustava korisniku. Programske komponente nazivaju se *artefakti* (engl. *artifacts*), dok se sklopovske komponente na kojima se one izvode nazivaju *čvorovi* (engl. *nodes*). S obzirom da se više čvorova može koristiti u implementaciji okoline za izvođenje programske potpore, oni se međusobno povezuju *vezama* (engl. *connections*).

Na slici 7.26 prikazan je dijagram razmještaja za primjer web aplikacije. Čvorovi koji predstavljaju fizičke sklopovske uređaje (*hardware*) na kojima se izvode programske komponente (artefakti) prikazani su kao trodimenzionalni kvadri, a za prikaz programskih artefakata koristi se simbol komponente kakav se koristi u dijagramu komponenata. Prikazana su klijentska računala (*Client workstation*) koja se spajaju na web aplikaciju preko web browsera koji se izvodi na njima kao programska komponenta. Klijenti se ne spajaju izravno na web server, već preko *load balancer servera* (koji koristi *nginx* serversku programsku komponentu) koji raspoređuje zahtjeve većeg broja korisnika ravnomjerno aplikacijskim serverima. Aplikacijski serveri izvode instance web aplikacije (*Web application*) i imaju dodatnu programsku komponentu za logiranje događaja (*Event logger*), a putem sigurne mreže odvojene od javne komuniciraju s fizički odvojenim serverom na kojem se nalazi baza podataka (koji koristi *MySQL* bazu podataka).

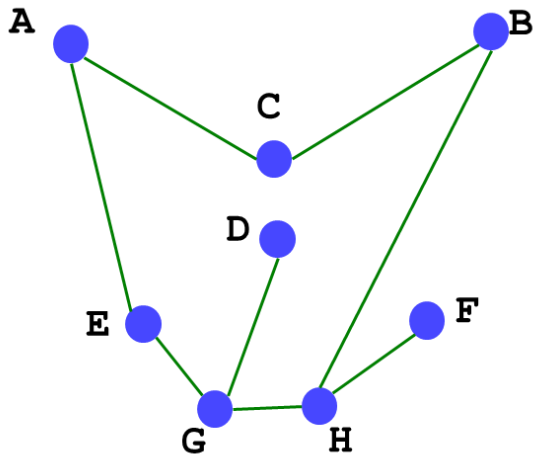


Slika 7.26: Dijagram razmještaja (primjer web aplikacije)

8 Grafovi

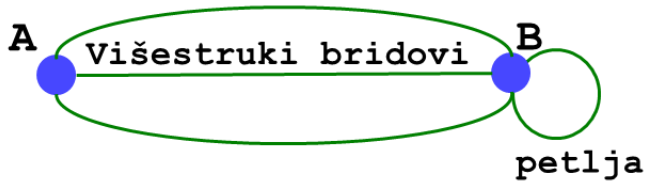
8.1 Osnovni pojmovi

Graf je struktura podataka (objekt) **G**={**V**, **E**} definirana na skupu točaka **V,** tzv. vrhova ili čvorova (engl. vertices) te spojnica E, tzv. bridova ili grana (engl. edges), među vrhovima **V**. Za neka dva vrha *u, v* ∈ **V** spojena s bridom *e* ∈ **E** kažemo da su vrhovi *u, v* incidentni s bridom *e*. Zatim vrhove *u, v* nazivamo krajevima brida *e* i međusobno susjednim vrhovima. Slika 8.1 prikazuje graf definiran na skupu točaka **V**={ A, B, C, D, E, F, H, H} povezanih skupom bridova **E**={ (A, E), (A, C), (C, B), (B, H), (D, G), (E, G), (G, H), (H, F)}. U konkretnom primjeru skupovi **V** i **E** su konačni te je time i dotičan graf konačan. Moguće je imati i graf bez bridova, tj. skup E=Ø je prazan skup, ali graf mora sadržavati barem jedna vrh. Graf s jednim vrhom nazivamo trivijalan graf.



Slika 8.1 Primjer grafa

Pravi brid (karika) je brid koji spaja različite vrhove dok brid koji spaja vrh s samim sobom nazivamo petljom. Ukoliko su dva ili više bridova spojena s istim parom vrhova tada govorimo o višestrukim bridovima.

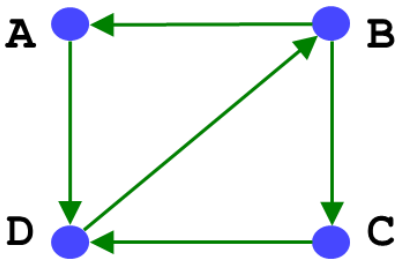


Slika 8.2 Graf s višestrukim bridovima između vrhova A i B; petlja u vrhu B

Osnovni parametri vezani uz konačne grafove su red grafa, veličina grafa te stupanj vrha grafa. Red grafa **G** predstavlja broj vrhova, v(**G**)=|**V**|. Veličina grafa **G** predstavlja broj bridova e(**G**)=|**E**|. Stupanj ili valencija $\mathbf{d}_G(v)$ nekog vrha *v* je broj bridova incidentnih (povezanih) s dotičnim vrhom *v*. Za ilustraciju, stupanj vrha grafa se može prikazati kao broj sjecišta kružnice oko vrha s bridovima vrha, uz napomenu da se petlja broji dva puta. Za vrh kažemo da je izoliran ukoliko vrijedi da je $\mathbf{d}_G(v)=0$, odnosno list ukoliko vrijedi $\mathbf{d}_G(v)=1$.

8.2 Vrste grafova

Jedna od najjednostavnijih vrsta grafova je neusmjereni (neorijentirani) graf kod kojeg bridovi nemaju smjer (orijentaciju). Drugim riječima brid *e*={A, B} ekvivalentan je bridu *e*={B, A}. Nasuprot tome usmjereni ili digraf (engl. directed graph) je onaj kod kojeg se bridovima pridružuje smjer te koji se obično tada nazivaju lukovima (Slika 8.3). Orijentirani graf (engl. oriented graph) je usmjereni graf gdje između dva vrha postoji samo jedan usmjereni brid (ponekad se u praksi orijentirani i usmjereni graf smatraju sinonimima). Kod grafova čijim bridovima je pridružen smjer stupnja ili valenciju vrha proširujemo na ulazni stupanj i izlazni stupanj. Prvo navedeni se odnosi na broj bridova (likova) koji ulaze u neki vrh, a drugo navedeni se odnosi na broj bridova (lukova) koji izlaze iz nekog vrha.



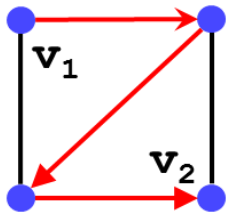
Slika 8.3 Usmjereni graf

Miješani graf (engl. mixed graph) je graf koji se sastoji od usmjerenih i neusmjerenih bridova. Ukoliko usmjereni graf ima višestruke bridove i petlje naziva se multidigraph (quier). Još šira generalizacija grafa ide prema hipergrafu gdje pojedini brid može povezivati više od dva vrha. Međutim u literaturi najčešće spominjani graf je jednostavan graf, odnosno neusmjereni graf bez višestrukih rubova i petlji. Ukoliko je svaki par vrhova jednostavnog grafa spojen brid tada ga nazivamo potpunim grafom. Pojava višestrukih bridova na neusmjerenom grafu definira multigraf, a dodatna pojava i petlji na takvom neusmjerenom grafu se naziva (ponekad) posebnim imenom, pseudograf. Bridovima grafa se mogu pridjeljivati različite brojčane vrijednosti, npr. udaljenosti, kapaciteti i sl.. Takve se vrijednosti za neku konkretnu primjenu tipično interpretiraju kao težine bridova, pa se tada graf naziva težinski graf.

8.3 Povezanost grafova, šetnje i putovi

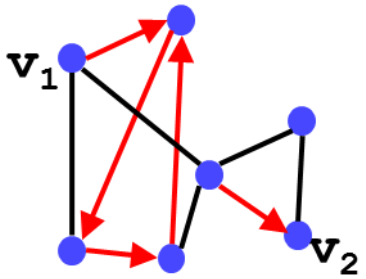
Neusmjereni graf je povezan (engl. *connected*) ako za svaki par vrhova postoji put između vrhova. Usmjereni graf je strogo povezan ako za svaki par vrhova postoji (usmjereni) put između vrhova. Ukoliko se na usmjerenom grafu uz zamjenu usmjerenih bridova s neusmjerenim dobije neusmjereni povezani grafa tada početni usmjereni graf nazivamo slabo povezanim grafom (engl. *weakly connected*).

Šetnja (engl. walk) je niz W=v0, e1, v1, e2 … ek-1, vk gdje su vi vrhovi a ei bridovi grafa takvi da za $1 \leq i \leq k$, brid ei povezuje vrhove vi i vi-1, a k se naziva dužina šetnje. Šetnja može proizvoljni puta proći nekim vrhom, odnosno bridom. Međutim šetnja koja ne uključuje niti jedan vrh dva puta (eventualno početni i krajnji vrh mogu biti isti) naziva se put (engl. path) (Slika 8.4). U djelu literature se već šetnja naziva putom, a put koji ne uključuje niti jedna vrh dva puta *elementarnim* putom.



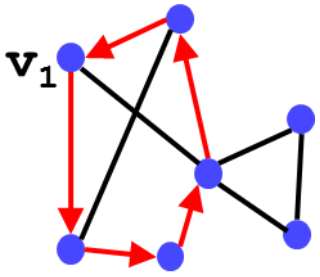
Slika 8.4 Put između vrhova v_1 i v_2 označen crvenom bojom i strelicama koje pokazuju redoslijed obilaska vrhova

Slično tome ukoliko šetnja ne prolazi dva puta istim bridom tada govorimo o stazi (engl. trail) (Slika 8.5).

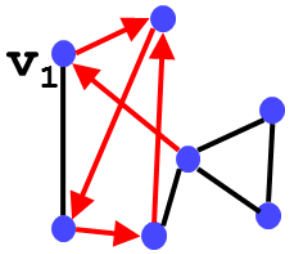


Slika 8.5 Staza između vrhova v_1 i v_2 označena crvenom bojom i strelicama koje pokazuju redoslijed obilaska vrhova

Staza može uključivati isti vrh dva puta, ako se primjerice dolazni i odlazni brid razlikuju (višestruki bridovi). Poseban slučaj puta koji započinje i završava u istom vrhu zovemo ciklus (engl. cycle) (Slika 8.6), a poseban slučaj staze koja završava i počinje u istom vrhu naziva se krug (engl. circuit) (Slika 8.7). Graf koji nema ciklusa se naziva aciklični graf (engl acyclic graph), dok se graf koji ima barem jedan ciklus naziva ciklični graf. (engl. cyclic graph). Usmjereni graf bez ciklusa naziva se usmjereni acikličan graf (engl. directed acyclic graph DAG).



Slika 8.6 Ciklus: put koji počinje i završava u vrhu v_1 te gdje je crvenom bojom i strelicama označen redoslijed obilaska vrhova.

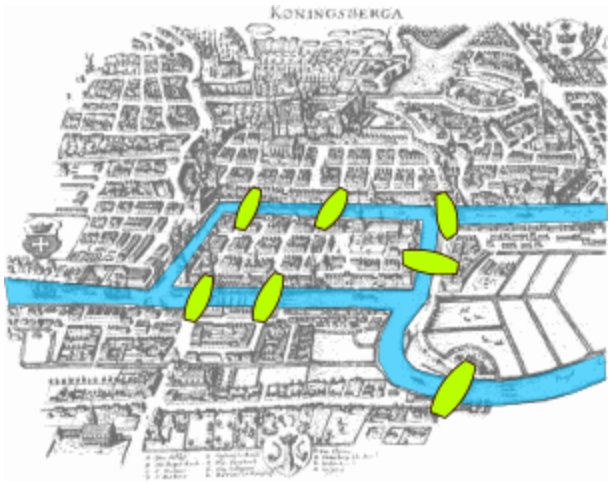


Slika 8.7 Krug koji počinje i završava u vrhu v_1 te gdje je crvenom bojom i strelicama označen redoslijed obilaska vrhova.

Za veliki broj algoritama nad grafovima od velikog je značaja pojam Hamiltonovog puta i Eulerove staze. Hamiltonov put se definira kao put kroz sve vrhove grafa (nije nužno proći i sve bridove grafa), te sukladno gornjim definicijama ako Hamiltonov puta započinje i završava u istom vrhu tada govorimo o Hamiltonovom ciklusu (ponekad zvanom i Hamiltonov krug). Eulerova staza (ponekad nazivana i put) je staza kroz sve bridove grafa gdje je također moguće proći isti vrh više puta. Eulerova staza kod koje su prvi i zadnji vrh jednaki nazivamo Eulerov ciklus (često zvan i Eulerov krug/tura). Graf koji ima Euleorv ciklus naziva se Eulerov graf.

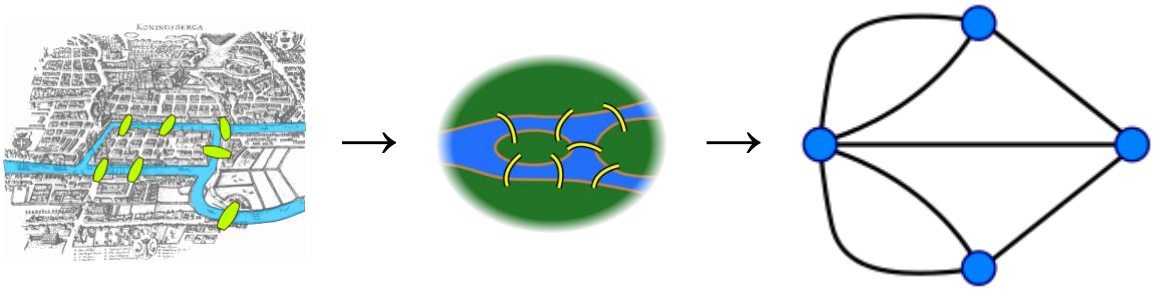
8.3.1 Primjena Eulerovog i Hamiltonovog puta

Koncept Eulerove staze ima interesantnu povijenu poveznicu s začetkom teorije grafova. Općenito se smatra da je povijesno gledajući teorija grafova započela početkom 18. stoljeća s Eulerovim rješenjem problema šetnje mostovima Königsberg-om [11], grada u Rusiji danas poznatog pod imenom Kaliningrad. Dijelovi grada bili su povezani s sedam mostova preko rijeke Pregola, a postavljeni problem je glasio da li je moguće prošetati gradom na način da se svaki most prođe jednom i samo jednom (Slika 8.8).



Slika 8.8 Mapa grada Königsberga (Kaliningrada) u Eulerovo vrijeme, s prikazom rasporeda sedam mostova preko rijeke Pregolae. Izvor [12].

Analizirajući problem Euler je različite dijelove grada označio vrhovima, a mostove bridovima te na taj način konstruirao graf (Slika 8.9).



Slika 8.9 Konstrukcija grafa preko mostova grada Königsberga i dijelova grada koji ti mostovi povezuju. Izvor [12].

Drugim riječima postavilo se pitanje postoji li na grafu Eulerova staza preko sedam bridova (mostova). Na posljetku je Euler zaključio da takva šetnja nije moguća jer se ne radi o tzv. Eulerovom grafu i što je kasnije formulirano kroz odgovarajući teorem: Povezani graf je Eulerov graf ako i samo ako mu je svaki vrh parnog stupnja. Povezani graf koji nije Eulerov (tj. nema Eulerov ciklus) može imati Eulerovu stazu ako i samo ako ima točno dva vrha neparnog stupnja. Graf konstruiran mostovima Königsberga ima očigledno četiri vrha s neparnim stupnjem (Slika 8.9), stoga nije moguće konstruirati niti Eulerovu ciklus, a čak niti barem stazu. Danas se problem prolaska bridovima grafa samo jednom, povezuje s tipom problema poznatim pod nazivom kineskog poštara (CPP**,** engl. Chinese Postman Problem), u

čast kineskog matematičara Mei-Ko Kwan (Meigu Guan) koji se bavio problemom i ponudio prva rješenja [13]. U sklopu takvog problema se ne traži bilo kakvo rješenje za Eulerov ciklus po grafu već se traži najkraći takav ciklus po grafu (relaksirani problem dozvoljava minimalan broj ponovljenih prelazaka preko istih rubova), odnosno poštar treba obići svaku kuću koristeći najkraću moguću rutu.

Vrlo slična grupa problema CPP-u je grupa problema ekvivalentna problemu trgovačkog putnika (engl. Travelling salesman problem). TPP postavlja uvjet na obilazak samo jednom vrhove grafa, a ne rubova grafa. Sukladno tome cilj više nije više pronaći Eulerov ciklus već Hamiltonov ciklus uslijed čega trgovački putnik treba proći svaki grad (vrh grafa) samo jednom i vratiti se, po mogućnosti, u početni grad. Naivno rješenje TPP problema koje bi isprobavalo sve moguće kombinacije obilaska vrhova grafa ima složenost O(n!) te postaje neuporabljivo za iole veći broj vrhova (npr. n>15). Nešto povoljniju složenost imaju algoritmi temeljeni na dinamičkom programiranju gdje se postiže složenost $O(2^n)$. Općenito govoreći egzaktno (optimalno) rješenje za TPP je teško pronaći u prihvatljivom vremenu. Zato se u praksi vrlo često pribjegava približnim rješenjima koja često garantiraju rješenje unutra određenih granica u odnosu na optimalno rješenje, a što se nerijetko pokazuje zadovoljavajući za praktične uporabu rješenje nekog problema tipa TPP-a.

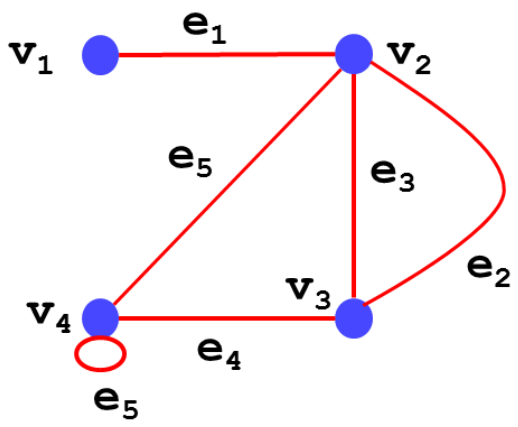
8.4 Reprezentacije grafova

Reprezentacija grafa grafički putem slike, odnosno preko vrhova povezanih bridovima, ima očiglednu prednost ilustrativnog prikaza. Međutim za efikasniju implementaciju algoritama nad grafovima koriste se alternativne reprezentacije grafova kao što je lista susjedstva (povezanosti), matrica susjedstva (povezanosti) i matrica incidencije.

8.4.1 Lista susjedstva

Za neki graf G={V, E} lista susjedstva (engl. adjacency list) je skup nesortiranih lista gdje svaka pojedina lista u skupu opisuje za neki vrh *v* s kojim je sve vrhovima povezan dotičan vrh. List susjedstva je moguće u kodu implementirati preko polja pokazivača A[|**V**|] gdje svaki element polja predstavlja jedan vrh i pokazuje na listu susjednih vrhova. Slika 8.10 i Tablica 3 prikazuju primjer grafa i njene reprezentacije listom susjedstva. Tablica 3 u

lijevom stupcu pokazuje pojedine elemente polja pokazivača, koji odgovaraju vrhovima grafova, te gdje vrijednosti pojedinih elemenata polja A[|**V**|] pokazuju na liste povezanosti dotičnog vrha s ostalim vrhovima (Tablica 3 desni stupac). U svakom čvoru pojedine liste moguće je zapisati i dodatne podatke kao što je težina brida. Jedna od prednosti prikaza listom susjedstva je efikasan memorijski zapis dok je nedostatak relativno dugi dohvat nekog element liste, zbog prolaska kroz listu.



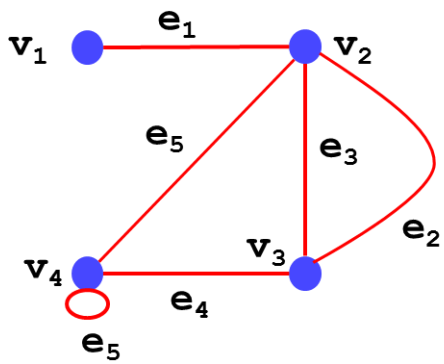
Slika 8.10 Primjer grafa prikazanog listom susjedstva na Tablica 3

Tablica 3 Lista susjedstva implementirana preko polja pokazivača grafa koji prikazuje Slika 8.10.

Lista susjedstva	
Polje pokazivača A[|V|]	Pojedinačne liste
v_1	$\rightarrow v_2$
v_2	$\rightarrow v_1 \rightarrow v_3 \rightarrow v_4$
v_3	$\rightarrow v_2 \rightarrow v_4$
v_4	$\rightarrow v_2 \rightarrow v_3 \rightarrow v_4$

8.4.2 Matrica susjedstva

Za neki graf G={**V**, **E**} te red grafa n=|**V**| matrica susjedstva (engl. adjacency matrix) je definirana kao kvadratna matrica $\mathbf{M_{nxn}}=[m_{ij}]$. Na taj način svakom se vrhu pridružuje redak *i* te stupac *j*. Vrijednost elementa m_{ij} =0 znači da vrh *i* s vrhom *j* nije povezan, tj. vrijednost različita od nule pokazuje s koliko bridova su vrhovi povezani. Slika 8.11 prikazuje primjer grafa za koji je izvedena matrica susjedstva **M** (36).



Slika 8.11 Primjer grafa prikazanog matricom susjedstva M (36)

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \tag{36}$$

Ukoliko graf nema petlji na dijagonalnim elementima biti će vrijednost nula, dok u konkretnom primjeru vrh v_4 ima petlju pa je element m_{4x4} =1. Uz pretpostavku izostanka petlji te prikaza neusmjerenog grafa, matrica susjedstva biti će simetrična. To omogućava vrlo kompaktan memorijski zapis. Neka se svaki element matrice zapisuje kao jedan bit tada je za zapis pola matrice **M** potrebno ~$|\mathbf{V}|^2/8/2$ byteova. Pogodnost matrice susjedstva očituje se u direktnom očitanju (ne)postojanje veze između dva vrha. Nadalje, suma retka (stupca) daje direktan odgovor na pitanje koliko bridova je incidentno s nekim vrhom. Primjerice, s vrhom v_4 (Slika 8.11) su incidentna tri bridova (suma četvrtog retka/stupca matrice **M** (36)). S druge strane ovakav zapis ne prikazuje odnose pojedinih vrhova s bridovima u kontekstu npr. da li je neki brid incidentan s dotičnim vrhom, ili da li je neki konkretan brid

ulazan/izlazan s dotičnim vrhom itd. Takve nedostatke moguće je umanjiti matricom incidencije.

8.4.3 Matrica incidencije

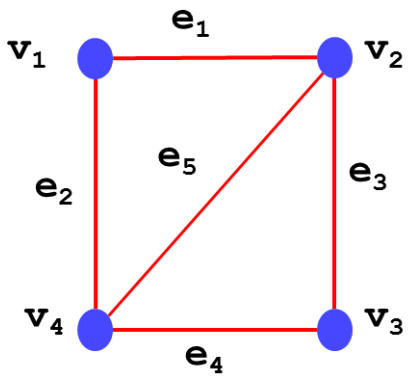
Neorijentirana matrica incidencije (engl. incidence matrix) za **G**={**V**, **E**} te n=|**V**| i m=|**E**|, neusmjerenog (neorijentiranog) grafa je definirana matricom $\mathbf{M}_{\mathbf{nxm}}=[m_{ij}]$, gdje su vrijednosti elemenata matrice definirane kao:

$$m_{ij} = \begin{cases} 1 & ako\ je\ brid\ j\ incidentan\ s\ vrhom\ i \\ 0 & inače \end{cases} \tag{37}$$

Na taj način svakom vrhu se pridružuje red *i* a svakom bridu stupac *j*. U slučaju postojanja petlje odgovarajući element iznosi m_{ij}=2. Izraz (38) pokazuje vrijednosti elemenata matrice incidencije prema grafu koji prikazuje Slika 8.12.

$$M = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{38}$$

Očigledno je odnos između pojedinih vrhova i rubova direktno čitljiv iz same matrice. Zatim suma redova odgovara valenciji $\mathbf{d}_G(v)$ nekog vrha *v*. Suma po stupcima iznosi dva budući da svaki brid ulazi i izlazi iz jednog vrha.



Slika 8.12 Primjer grafa prikazanog matricom incidencije M (38**)**

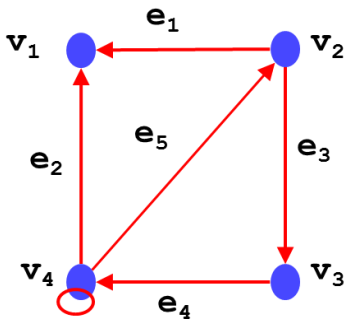
Gornju reprezentaciju moguće je modificirati u slučaju usmjerenog grafa. Orijentirana matrica incidencije (engl. incidence matrix) za **G**={**V**, **E**} te n=|**V**| i m=|**E**|, usmjerenog (orijentiranog) grafa je definirana matricom $\mathbf{M}_{\mathbf{nxm}}=[m_{ij}]$, gdje su vrijednosti elemenata matrice definirane kao:

$$m_{ij} = \begin{cases} 1 & ako\ brid\ j\ izlazi\ iz\ vrha\ i \\ 0 & ako\ brid\ j\ i\ vrh\ i\ nisu\ incidentni \\ -1 & ako\ brid\ j\ ulazi\ u\ vrh\ i \end{cases} \tag{39}$$

Npr., ako se neusmjereni graf koji prikazuje Slika 8.12 pretvori u usmjereni tako da se proizvoljno dodijele smjerovi rubovima dobivamo graf prikazan na Slika 8.13, tada izraz (40) pokazuje vrijednosti elemenata matrice incidencije sukladno usmjerenom grafu.

$$M = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 & 1 \end{bmatrix} \tag{40}$$

U tome slučaju suma po stupcima odgovara nuli, a suma po recima apsolutnih vrijednosti elementa m_{ij} odgovara valenciji nekog vrha v. Pojavu petlje je uobičajeno označavati s nulom.



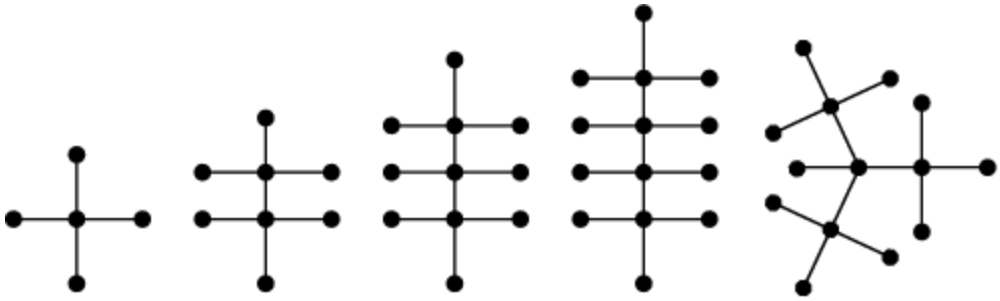
Slika 8.13 Primjer usmjerenog grafa prikazanog matricom incidencije M (40)

8.5 Poveznica između grafa i stabla

Slično kao što postoje generalizacije grafa, npr. hipergraf (pojedini brid može povezivati više od dva vrha), moguće je ići i u suprotnome smjeru gdje se definiraju pojedini specijalni oblici grafa. Jadan od takvih najpopularnijih oblika je stablo. Graf **G**={**V**, E} koji zadovoljava bilo koji od sljedećih uvjeta nazivamo stablom:

- **G** ima **V**-1 bridova i nema ciklusa (acikličan graf)
- **G** ima **V**-1 bridova i **G** je povezani graf
- točno jedan put povezuje svaki par vrhova u G
- **G** je povezan graf, ali prestaje biti ukoliko se bilo koji od bridova ukloni

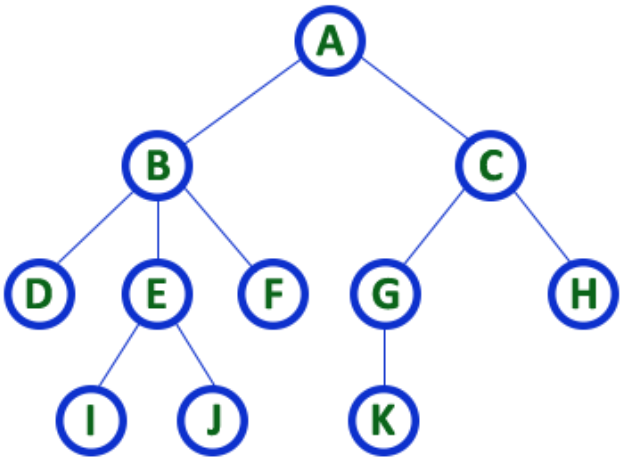
Tablica 4 pokazuje usporedno najvažnija razlike i sličnosti između stabla i grafa.



Slika 8.14 Primjer stabala. Izvor [14].

Slika 8.14 prikazuje nekoliko primjera mogućih izgleda stabala. Skupina odvojenih stabala predstavlja šumu (engl. forest). Poseban slučaj stabla je korjenovano stablo (engl. Rooted tree) gdje se jednom od vrhova stabla dodjeli poseban status tzv. korijena stabala (Slika 8.15). Korijen u kontekstu stabla kao strukture podataka, predstavlja izvorište, tj. prvi vrh preko kojega se pristupa svim ostalima. Na taj način se korjenovanjem stabla posredno dodjeljuje smjer bridovima stabla, a što ima sljedeće posljedice. Smjer bridova između korijena ide od vrha bliže korijenu k vrhu dalje od korijena. Korijen je vrh koji ima ulazni stupanj (broj bridova koji ulaze u vrh) nula. Ostali vrhovi imaju ulazni stupanj jedan. Vrhovi koji imaju izlazni stupanj nula nazivaju se listovi stabla. Za neka dva susjedna vrha, vrh bliži korijenu naziva se roditeljem (engl. parent), a onaj dalje od korijena naziva se djetetom (engl. child.). Daljnja specijalizacija stabla može biti binarno stablo gdje svaki vrh ima najviše

dvoje djece (Slika 8.15). Nadalje, takvo binarno stablo može biti uređeno po određenom principu, pa se naziva binarnim stablom za pretraživanje (engl. binary search tree).



Slika 8.15 Korjenovano stablo: vrh A predstavlja korijen; vrhovi D, I, J, F, K i G su listovi stabla. Jedino vrh B ima više od dvoje djece. Izostankom nekog od vrhova D ili F prikazano stablo poslao bi binarno stablo.

Tablica 4 Usporedni prikaz najvažnijih razlika i sličnosti stabla i grafa.

	Stablo	**Graf**
Put	Između svaka dva čvora postoji samo jedan put	Može postojati više putova između različitih vrhova.
Petlja	Ne postoji petlja (engl. cycle) niti krug (engl. circuit). Time je složenost manja	Može imati i petlje i krugove s imati složeniju strukturu.
Korijen	Moguće je jednom vrhu dodijeliti atribut 'korijen' čime se definira stablo s korijenom (engl. rooted tree).	Ne postoji pojam korijena.
Roditelj-dijete veza (engl. parent child relationship)	Susjedno povezani vrhovi definiraju odnos roditelj-dijete čime se definira smjer prolaska stablom od vrha prema dnu ili od dna prema vrhu	
Obilazak vrhova	Uobičajeni načini obilaska su poznati pod imenima Pre-Order, In-Order te Post-Order.	Uobičajeni načini obilaska su poznati pod imenima Depth First Search i Breadth First Search
Uobičajene vrste	binarno stablo (engl. binary tree), binarno stablo za pretraživanje (engl. binary search tree), gomila (engl. heap)	Osnovna podjela je na usmjerene i neusmjereni grafovi.
Struktura	hijerarhijska	mrežna

8.6 Obilazak grafa

Postoji više načina za prolaz vrhovima grafa, a gdje nije primarno ispuniti neki dodatni kriterij optimizacije (npr. proći najkraći put između vrhova) već samo sustavno posjetiti sve vrhove grafa. Dva najpopularnija načina su obilazak po dubini (engl. depth first search DFS) i obilazak po širini (engl. breadth first search BFS). Interesantno je istaknuti da i DFS i BFS daju i neke informacije o strukturi grafa, npr. povezanosti.

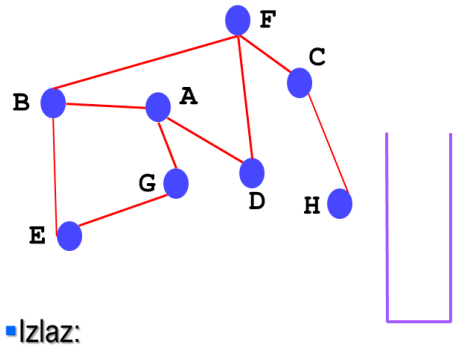
8.6.1 DFS obilazak vrhova grafa

Za neki zadani početni vrh, DFS obilazak se temelji na principu obilaženja najudaljenijih susjeda vrhova tako što se ide što više u dubinu (što dalje od početnog vrha), slijedeći rubove između vrhova. Svaki puta kada se dosegne vrh grafa koji više nije s niti jednim drugim vrhom povezan, tada se ode korak unatrag (engl. backtracking) te se kreće od još neposjećenog vrha ponovno u dubinu. Ključna stvar kod DFS obilaska je izbjeći obilazak (posjećivanje) već obiđenih vrhova, a što se relativno efikasno može programski implementirati korištenjem strukture podatka stog. Kompletan DFS algoritam može se sažeti u nekoliko koraka:

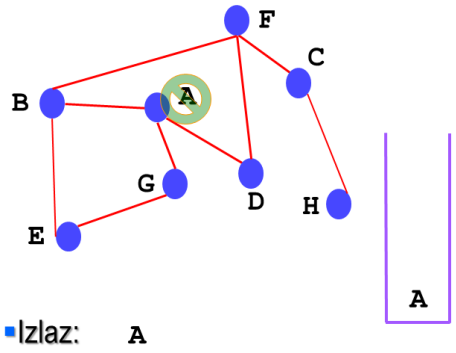
- kreirati prazan stog **S**
- odabrati početni vrh, označiti ga ‘posjećenim’ te ga dodati (stog operacija push) na stog **S** i na izlaz (npr. print)
- dokle god stog **S** nije prazan ponavljati sljedeće:
    - pogledati (stog operacija peek) vrh *u* s stoga **S**
    - ako vrh *u* ima neoznačeni susjedni vrh *v* tada:
        - označiti vrh v ‘posjećenim’ te ga dodati na stog S i na izlaz (print)
    - inače:
        - maknuti (stog operacija pop) vrh *u* s stoga **S**

Slijedeći niz slika za zadani primjer grafa sastavljenog od vrhova **V**={A, B, C, D, E, F, G, H} (Slika 8.16) pokazuje detaljnije pojedine korake DFS obilaska. Pretpostavimo da je na početku odabran vrh *A* kojeg označavamo kao posjećenog, stavljamo ga na stoga te šaljemo na izlaz (Slika 8.17). Nakon toga, sukladno gore opisanom postupku, pogledamo vrh stoga te

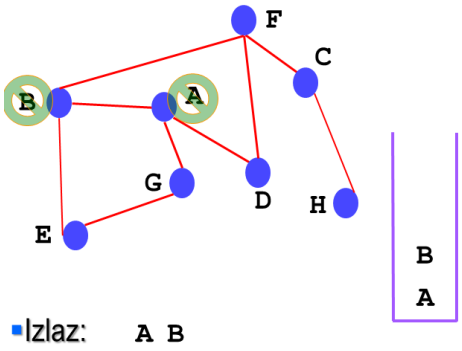
odabiremo za obići nekog od susjeda vrha *A*, primjerice vrh *B*. Odabir susjeda je u općenitom smislu proizvoljan, a za konkretna primjer se neposjećeni susjedi odabiru prema abecednom rasporedu naziva vrhova. Vrh *B* također označavamo posjećenim dodajemo na stoga i šaljemo na izlaz (Slika 8.18). Sada jer vrh *B* na vrhu stoga te za sljedeći vrh odabiremo neposjećeni vrh susjedan vrhu *B*, a što po abecednom redoslijedu proizlazi da je vrh *E*. Za vrh *E* ponavljamo iste radnje kao i za vrhove A i B: označavanje da je posjećen, stavljanje na stog te slanje na izlaz (Slika 8.19). U slijedećem koraku imamo samo jedan izbor za odabir neposjećenog vrha, susjednog vrhu *E*, a to je vrh *G* koji obrađujemo kao i sve dosadašnje vrhove (Slika 8.20). U konkretnom primjeru ovo je sada prvi puta situacija da za neki vrh grafa, konkretno vrh G, nema više neposjećenih susjeda te DFS postupka nalaže da maknemo vrh *G* s stoga (Slika 8.21). Štoviše, slična situacija se ponavlja i za slijedeći vrh s vrha stoga, a to je vrh *E*, pa i njega mičemo s stoga (Slika 8.22). Nakon toga se na vrhu stoga nalazi vrh *B* koji od tri susjedna vrha ima još jednog neposjećenog susjeda, vrh *F* (Slika 8.23). Ostatak postupka (Slika 8.24, Slika 8.25, Slika 8.26, Slika 8.27) za preostale vrhove je analogan već objašnjenim koracima. Neposredan kraj DFS obilaska označava uklanjanje svih vrhova s stoga (Slika 8.28).



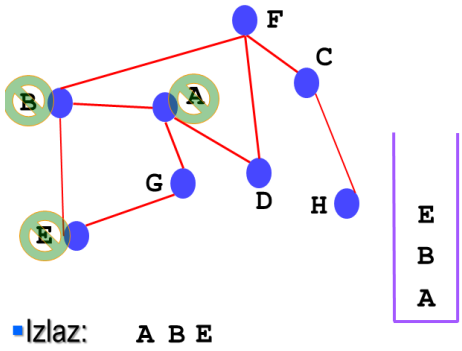
Slika 8.16 Primjer grafa za DFS obilazak. Desni ljubičasti okvir predstavlja strukturu podataka stog



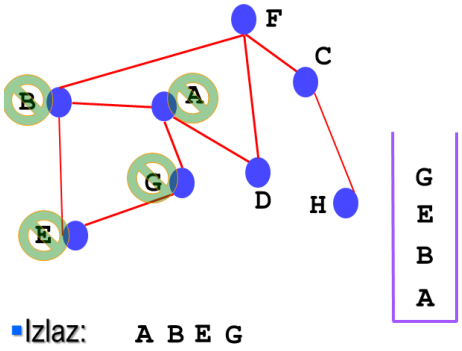
Slika 8.17 Posjećivanje, dodavanje na stoga i slanje na izlaz vrha A.



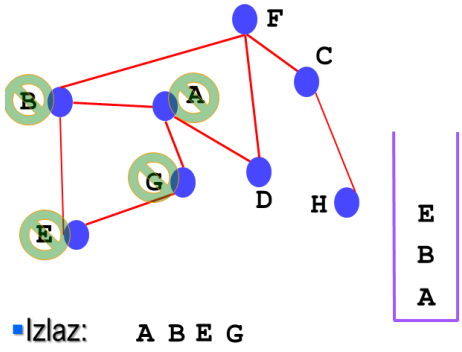
Slika 8.18 Posjećivanje, dodavanje na stoga i slanje na izlaz vrha B.



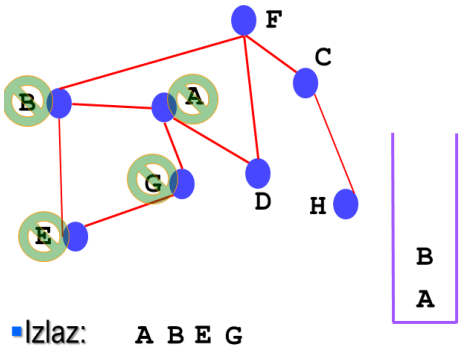
Slika 8.19 Posjećivanje, dodavanje na stoga i slanje na izlaz vrha E.



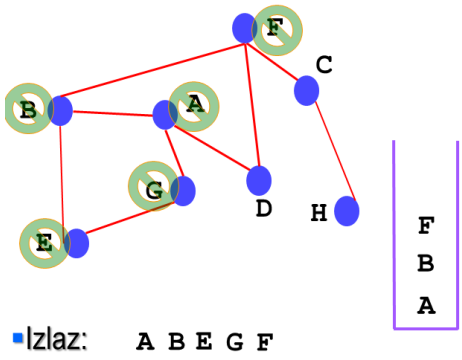
Slika 8.20 Posjećivanje, dodavanje na stoga i slanje na izlaz vrha G.



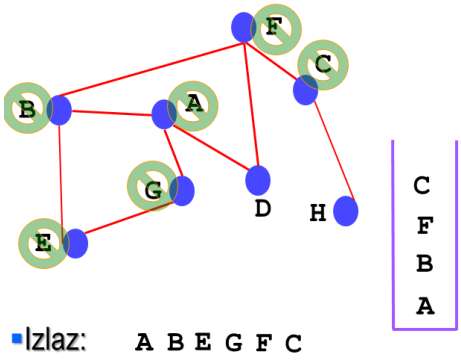
Slika 8.21 Uklanjanje vrha G s stoga.



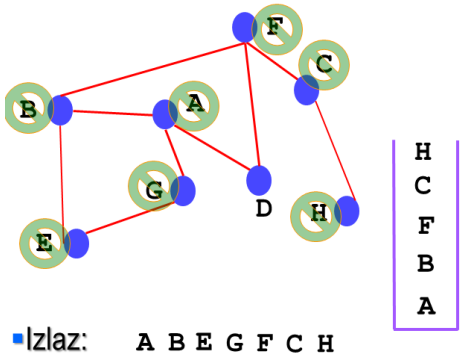
Slika 8.22 Uklanjanje vrha E s stoga



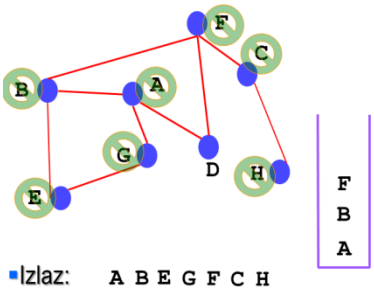
Slika 8.23 Posjećivanje, dodavanje na stoga i slanje na izlaz vrha F.



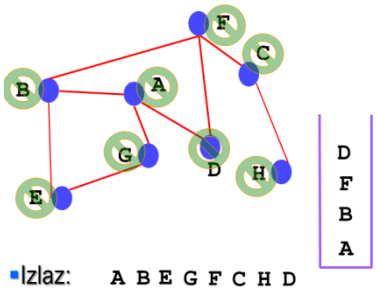
Slika 8.24 Posjećivanje, dodavanje na stoga i slanje na izlaz vrha C.



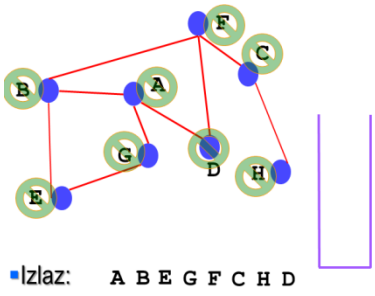
Slika 8.25 Posjećivanje, dodavanje na stoga i slanje na izlaz vrha H.



Slika 8.26 Uklanjanje s stoga vrhova C i H koji više nemaju neposjećenih susjeda.



Slika 8.27 Posjećivanje, dodavanje na stoga i slanje na izlaz posljednjeg vrha, vrha D.



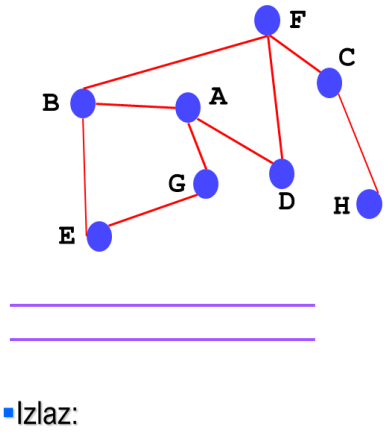
Slika 8.28 Nema više neposjećenih vrhova. Uklanjanje s stoga svih vrhova. Kraj DFS postupka.

8.6.2 BFS obilazak vrhova grafa

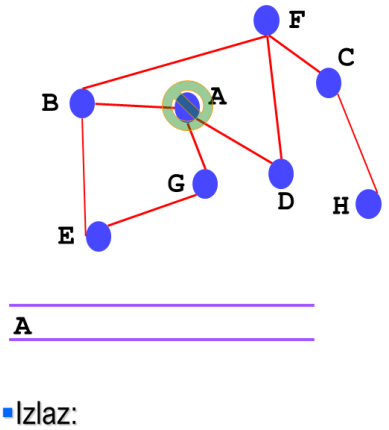
Za neki početni (razmatrani) vrh, BFS obilazak se temelji na principu posjećivanja najbližih susjednih vrhova u odnosu na razmatrani vrh. Kada se posjeti cjelokupno susjedstvo početnog vrha tada se svaki od obiđenih susjeda uzima za razmatrani vrh te se obilaze još neobiđeni svi najbliži susjedni vrhovi. Postupak se analogno ponavlja za zadnje posjećene vrhove u posljednjem koraku. Na taj način obiđeno područje grafa ima tendenciju povećanja u širinu, u odnosu na početno odabrani vrh. Slično kao i kod DFS-a ovdje također treba izbjeći posjećivanje već posjećenih vrhova. Međutim kod BFS-a programska struktura. Kompletan BFS algoritam može se sažeti u nekoliko koraka:

- kreirati prazni red Q,
- odabrati početni vrh, označiti ga 'posjećenim' te dodati u red Q
- dokle god red Q nije prazan
    - skinuti vrh *u* na izlazu iz reda Q te ga dodati na izlaz (print)
    - za svaki *v* susjedan vrhu *u*
        - ako *v* nije bio posjećen tada ga označiti kao posjećenog i dodati u red Q

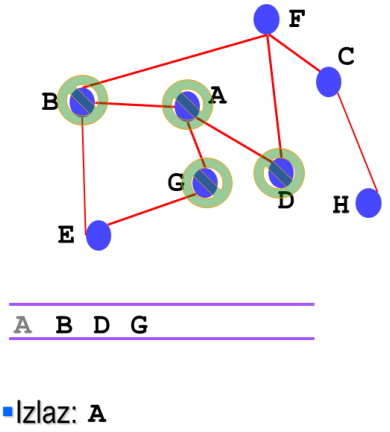
Slijedeći niz slika za zadani primjer grafa sastavljenog od vrhova **V**={A, B, C, D, E, F, G, H} (Slika 8.29), pokazuje detaljnije pojedine korake BFS obilaska. Pretpostavimo da je na samome početku odbran vrh A kojeg označavamo posjećenim i dodajemo u red (Slika 8.30). BFS postupak dalje nalaže uzimanje prvog vrha na izlazu iz reda, tj. vrha A, slanje vrha A na izlaz, te stavljanje svih neposjećenih susjeda u red te njihovo označavanje posjećenima (Slika 8.31). Možemo pretpostaviti stavljanje neposjećenih susjeda u red abecednim redoslijedom. Nakon toga prvi dostupan vrh na izlazu reda je vrh B koji šaljemo na izlaz, a njegove susjede stavljamo u red i označavamo posjećenima (Slika 8.32). Na taj način prvi dostupan vrh za uklanjanje iz reda postaje vrh D koji nema neposjećenih susjeda te obrada vezna uz njega se odnosi samo na njegovo uklanjanje iz reda i slanje na izlaz. Slično vrijedi i za sljedeća dva vrha u redu: G i E (Slika 8.33). Za preostala tri vrha F, C i H vrijedi analogan postupak (Slika 8.34, Slika 8.35). a BSF postupak završava uklanjanjem posljednjeg vrha iz reda, vrha H (Slika 8.36).



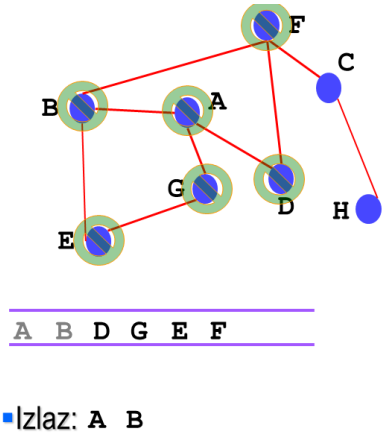
Slika 8.29 Primjer grafa za BFS obilazak. Ljubičasti okvir predstavlja strukturu podataka red.



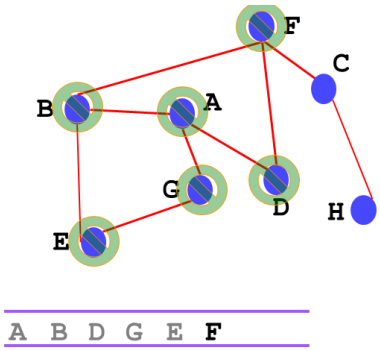
Slika 8.30 Posjećivanje i dodavanje u red vrha A.



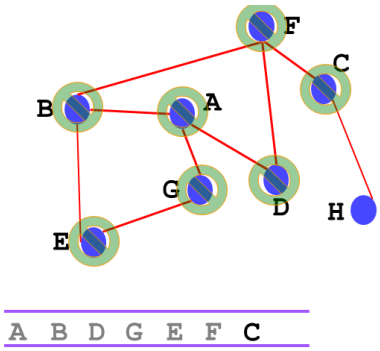
Slika 8.31 Uklanjanje vrha A iz reda te slanje na izlaz. Stavljanje u red i označavanje posjećenim svih do sada neposjećenih susjeda vrha A: B, D i G.



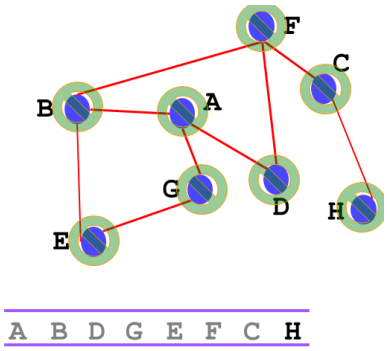
Slika 8.32 Uklanjanje vrha B iz reda te slanje na izlaz. Stavljanje u red i označavanje posjećenim svih do sada neposjećenih susjeda vrha B: E i F.



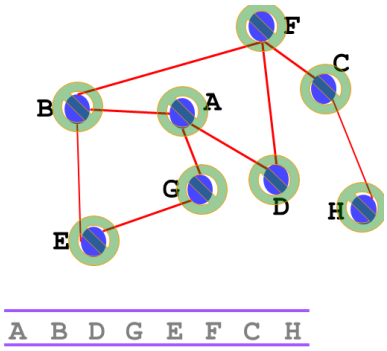
Slika 8.33 Uklanjanje vrhova D, G i E iz reda te slanje istih na izlaz. Nitko od dotična tri vrha nema više neposjećenih susjeda za stavit u red.



Slika 8.34 Uklanjanje vrha F iz reda te slanje na izlaz. Stavljanje u red i označavanje posjećenim svih do sada neposjećenih susjeda vrha F: C.



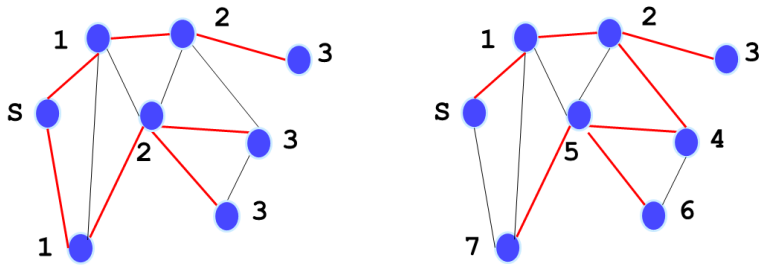
Slika 8.35 Uklanjanje vrha C iz reda te slanje na izlaz. Stavljanje u red i označavanje posjećenim svih do sada neposjećenih susjeda vrha C: H.



Slika 8.36 Uklanjanje vrha H iz reda te slanje na izlaz. Vrh H nema neposjećenih susjeda. Red je ispražnjen, kraj BFS postupka.

8.6.3 Razapinjuće stablo

Stablo **T** je razapinjuće stablo grafa **G** ukoliko stablo **T** sadrži sve vrhove grafa **G**. **T** je u biti podgraf grafa **G** koji sadrži sve vrhove grafa **G**. Interesantno je primijetiti da DFS i BFS algoritmi posjećuju sve vrhove grafa dok prolaze kroz samo jedan dio rubova. Taj skup prođenih rubova i posjećenih vrhova čini upravo razapinjuće stablo grafa. Slika 8.35 prikazuje primjer razapinjućih stabala primjenom BSF i DSF algoritama. 'S' je odabran za početni vrh, a brojevi kod ostalih vrhova predstavljaju redoslijed obilazaka. Unutar skupine vrhova označenih s istim brojem je redoslijed, u općenitom slučaju, proizvoljan (BSF algoritam).



a) BFS razapinjuće stablo b) DFS razapinjuće stablo

Slika 6.20 Primjeri BSF i DSF obilazak grafa. Obiđeni rubovi, označeni crvenom bojom, predstavljaju razapinjuća stabla.

Također i za DSF algoritam obilaska moguće je, u općenitom slučaju, dobiti različita razapinjuća stabla. Primjerice počevši od vrha **S** (Slika 8.35 b)), za prvi sljedeći vrh mogli smo odabrati vrh označen kao '7' umjesto vrha označenog s '1', a što bi u konačnici dalo razapinjuće stablo drugačijeg izgleda. Ponekad je cilj pronaći razapinjuće stablo težinskog (ne) usmjerenog grafa čija suma odabranih rubova je minimalna. Takvo stablo se naziva minimalno razapinjuće stablo (engl. minimum spanning tree MST). Postoje više od jednog algoritma za pronalazak MST, a jedna od popularnijih je Prim's algoritam [15].

8.7 Topološki sort

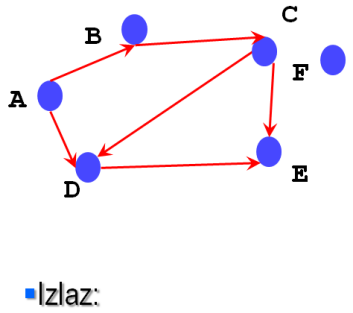
Poznato je da usmjereni ciklički graf (DAG) po definiciji nema (usmjerenih) ciklusa, odnosno idući od jednog od vrhova v_i, sljedeći usmjereni rub, prema nekom drugom vrhu v_j, pa zatim prema sljedećem vrhu v_k itd. nije moguće vratiti se u početni vrh v_i. Međutim grafovi takvog tipa imaju tzv. topološki redoslijed (engl. topological ordering). To znači da postoji takav niz (redoslijed) vrhova grafa gdje u kojem je redoslijed susjednih vrhova uvjetovan usmjerenim rubovima kojih ih povezuju. Topološki redoslijed nalazimo topološkim sortom čiji je znači zadatak pronaći za usmjereni graf G bez ciklusa takav poredak vrhova gdje će za svaki usmjereni brid (u, v) vrijediti da će u poretku vrh u doći prije vrha v. Ukoliko graf nije DAG topološko sortiranje nije moguće. Potreba za topološkim sortiranjem se u praksi tipično javlja kada vrhovi grafa predstavljaju zadatke (npr. predmeti studija za položiti), a usmjerenost bridova između vrhova definira uvjete koji ukazuju na nužnost redoslijed izvođenja zadataka (npr. redoslijed polaganja/upisa predmeta studija). Spomenimo da je pojam sortiranje vrlo često vezan za dovođenje u poredak određenih entiteta temeljem numeričkih vrijednosti. Međutim topološki sort je primjer tzv. ne-numeričkog sort gdje kriterij za poredak nisu numeričke vrijednosti (vrhova) već usmjerenost bridova između pojedinih vrhova.

Popularan algoritam za topološko sortiranje je Kahn-ov algoritam, originalno objavljen u [16]. U suštini Kahn-ov algoritam se temelji na jednostavnoj zapažanju da onaj vrh koji nema ulaznih rubova, tj. onaj zadatak koji nema za preduvjet izvršenje nekog prethodnog zadatka, takav zadatak se može izvršiti odnosno takav vrh se može (na)dodati u predviđeni topološki redoslijed. U biti Kahn-ov algoritam prolazi kroz graf i bira upravo takve vrhove koji nemaju ulaznih bridova te ih dodaje u topološki redoslijed. Inače postojanje barem jednog takvog vrha u grafu je uvjet da bi graf bio acikličan. Cjelokupan Kahn-ov algoritma može se sažeti u nekoliko koraka:

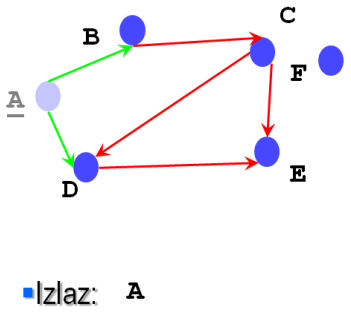
- a) identificiraj vrhove koji nemaju ulaznih bridova (startni vrhovi) te odaberi jedan startni vrh
- b) stavi startni vrh u izlazni red

- c) ukloni sve izlazeće bridove startnog vrha, tj. ulazne bridove vrhova s kojima je povezan startni vrh
- d) ponavljaj prethodne korake do praznog grafa

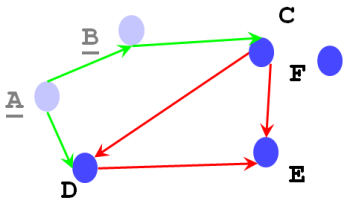
Slika 8.37 je primjer DAG grafa. Za primjenu Kahn-ovo algoritma očigledno postoje dva vrha koji nemaju ulazne bridovoe, a to su vrhovi A i F. Primjećuje se da očigledno topografsko sortiranje ne daje nužno jedinstveno rješenje. Pretpostavimo da je odabran vrh A te uklonimo sve izlazne bridove iz vrha A (Slika 8.38). Uklanjanjem izlaznih bridova vrha A ponovno imamo za izbor odabrati dva vrha bez ulaznih bridova, a to su vrhovi B i F. Odaberimo vrh B te ponovimo postupak isti kao i za prethodni vrh A (Slika 8.39). Nakon ovoga vrha, prilikom izbora odabira vrh C ili vrh F, proizvoljno odaberimo vrh F, koji inače nema niti ulaznih niti izlaznih bridova, pa nakon što ga označimo kao odabranog samo ga još svrstamo u izlazni red (Slika 8.40). Analogan postupka Kahn-ovog algoritma nastavljamo s vrhovima C i D (Slika 8.41 Slika 8.42). Konačno topološko sortiranje završava s vrhom E (Slika 8.43).



Slika 8.37 Odmah na početku postoje dva moguća startna vrha, vrhovi A i F.

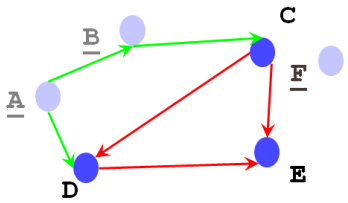


Slika 8.38 Označavamo odabrani vrh A, uklanjamo iz grafa sve njegove izlazne bridove (označeno zelenim) te ga stavljamo u izlazni red.



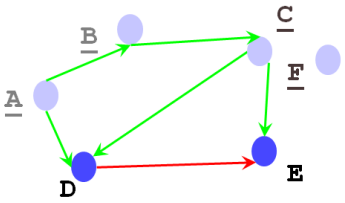
▪Izlaz: A B

Slika 8.39 Označavamo odabrani vrh B, uklanjamo iz grafa sve njegove izlazne bridove (označeno zelenim) te ga stavljamo u izlazni red.



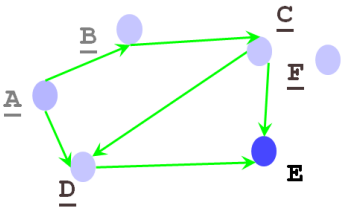
▪Izlaz: A B F

Slika 8.40 Označavamo odabrani vrh F, koji nema izlaznih bridova za ukloniti te ga još samo stavljamo u izlazni red.



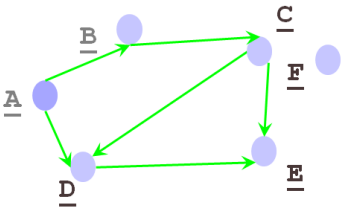
▪Izlaz: A B F C

Slika 8.41 Označavamo odabrani vrh C, uklanjamo iz grafa sve njegove izlazne bridove (označeno zelenim) te ga stavljamo u izlazni red.



▪Izlaz: A B F C D

Slika 8.42 Označavamo odabrani vrh D, uklanjamo iz grafa sve njegove izlazne bridove (označeno zelenim) te ga stavljamo u izlazni red.



▪Izlaz: A B F C D E

Slika 8.43 Označavamo još vrh E, kao jedini preostali vrh u grafu i to bez izlaznih bridova, te ga za sami kraj još samo stavljamo u izlazni red.

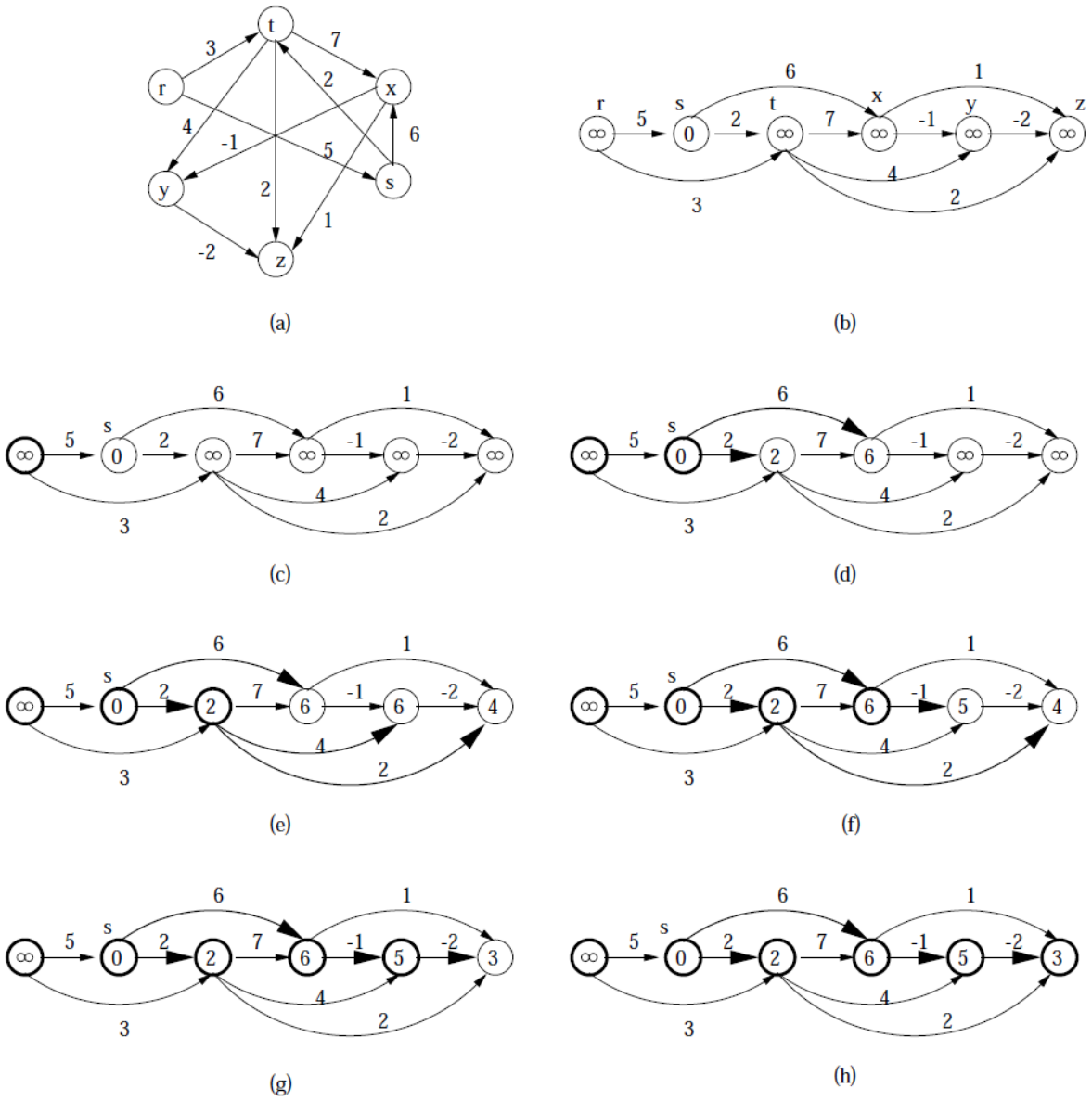
8.7.1 Nalaženje najkraćeg puta između vrhova grafa

Česta potreba u praksi je pronaći najkraći put između dva vrha nekog težinskog grafa. U kontekstu težinskog grafa najkraći put uobičajeno pretpostavlja put koji daje minimalnu sumu težina, tj. cijenu puta, između neka dva vrha na grafu. Radi li se o DAG tipu grafa, rezultat topološkog sorta je moguće iskoristiti u tu svrhu. Topološki sort za neki vrh daje nužne redoslijede koji se moraju poštivati da bi se iz nekog vrha *s* došlo do bilo kojeg od ostalih vrhova grafa. Ideja sljedećeg algoritma je za neki početni vrh *s* odabrati put prema susjednim vrhovima koji daje najmanju cijenu, uz poštivanje obilaska susjeda sukladno topološkom sortu. Budući da uslijed složenosti veza između vrhova grafa do istog susjeda je moguće doći na više načina, svaki put se neka nova alternativa puta uspoređuje s trenutno zapamćenom cjenovno najjeftinijom alternativom. U tu svrhu za poznati topološki poredak vrhova u polju **V** grafa **G** te odbrani vrh *s* algoritam glasi:

- Inicijalizacija:
    - Definira se polje **d**, veličine kao i **V**, u koje će se zapisati najkraće udaljenosti od odabranog vrha *s* prema ostalim vrhovima. Inicijalizacija: **d**[*s*] = 0, ostali vrhovi u **d**[*u*] = ∞.
    - Definira se i polje **p**, polje veličine kao i **V**, u koje će se za svaki vrh zapisati njegov prethodnik na dijelu najkraćeg puta. Inicijalizacija: za sve elemente **p** = NULL.
- ponavljaj za sve vrhove *u* topološki poredane u **V**, počevši od vrha *s*
    - za svaki vrh *v* koji slijedi nakon vrha *u* (tj. postoji brid (*u*, *v*))
        - neka je *w* težina brida (*u*, *v*)
        - if d[v] > d[u] + w, tada
            - d[v] ← d[u] + w
            - p[v] ← u

Gornji algoritma je moguće pokazati na sljedećem primjeru(Slika 8.44). Slika 8.44 a) pokazuje početni graf, a topološki sortirana struktura grafa dana je na Slika 8.44 b). Vrh označen kao *s* je početni vrh te sukladno inicijalizacijskom djelu s početka algoritma udaljenosti prema svim ostalim vrhovima se postavljaju na beskonačno, udaljenost od samog

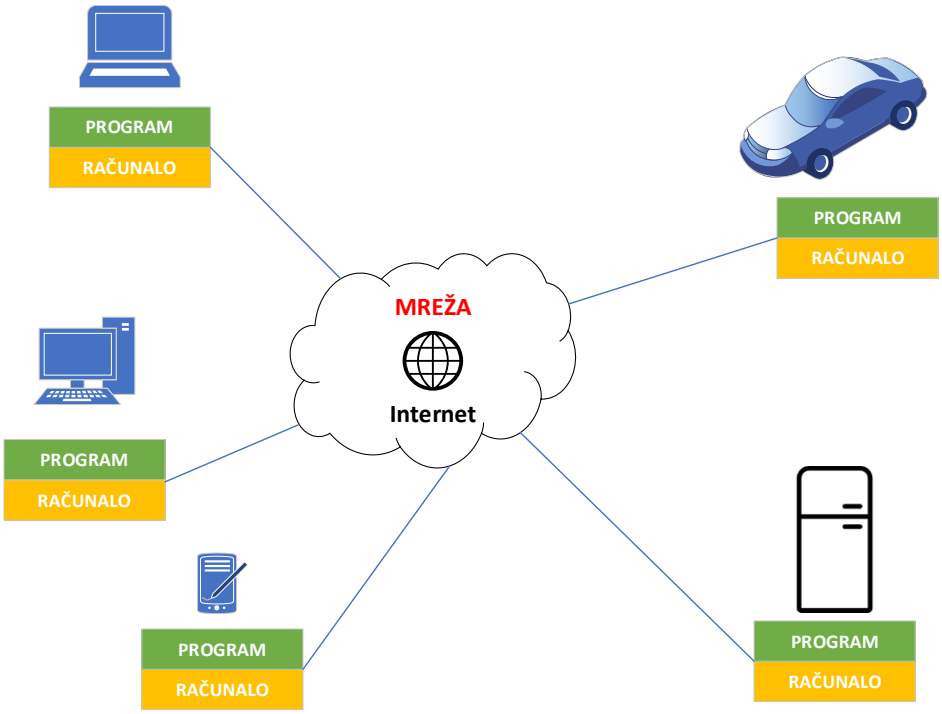
vrha *s* na nulu (Slika 8.44 b). Trenutno razmatrani vrh topološkog redoslijeda je na sljedećim slikama označen podebljanom linijom. Prvo razmatrani vrh je vrh *r*. Budući da je odabrani početni vrh *s* u topološkom redoslijedu se nalazi nakon vrha *r*, tj. do njega nije moguće doći, udaljenost prema vrhu *r* ostaje beskonačno, d[r]=∞ (Slika 8.44 c)). Slijedi sama bit algoritma. Razmatramo sljedeći vrh topološkog redoslijeda, a to je upravo odabrani početni vrh *s*, za kojeg vidimo da je moguće doći do vrhova *t* i *x* po cijeni (težini bridova) 2, odnosno 6. S obzirom da prema nejednakostima gornjeg algoritma za oba vrha vrijedi **d**[v=t]=∞>**d**[u=s]+w=0+2, odnosno **d**[v=x]=∞>**d**[u=s]+w=0+6, obnavljamo vrijednosti u polju **d** kao **d**[v=t]=2, tj. **d**[v=t]=6 (Slika 8.44 d)). Također pratimo i sami obilazak najkraćeg puta u polju **p** kao **p**[x]=s te **p**[t]=s. U novoj iteraciji algoritma razmatramo novi vrh prema topološkom redoslijedu, a to je vrh *t*, iz kojega možemo bridovima dosegnuti vrhove *x*, *y* i *z*. Slično kao i raije, zapažamo da cijena dolaska u vrhove y i z iz vrha s preko t je manja od trenutno zapamćene minimalne cijene (beskonačno) te obnavljamo vrijednosti u polju **d** s novom minimalnom cijenom, tj. zapis puta u polju **p**. Međutim analiza dolaska iz vrha *t* u vrh *x* preko brida čija je težina w=7 bi povećao bi ukupnu cijenu dolaska iz početnog vrha *s* u vrh *x* (koja iznosi 6) te odustajemo od obnavljanja vrijednosti polja **d** i **p** (Slika 8.44 e)). Drugim riječima kraći (jeftiniji) put je doći u vrh *x* direktno iz vrha *s* nego preko vrha *t*. Zatim prema topološkom redoslijedu razmatramo vrh *x* iz kojeg bridovima možemo doći u vrhove *y* i *z*. S obzirom na težine korespondentnih bridova koje povezuju vrh *x* s vrhovima *y* i *z* te trenutnih cijena od početnog vrha *s* prema vrhovima *y* i *z*, zaključujemo da će nam dolazak u vrh *y* preko razmatranog vrha *x* donijeti manju ukupnu cijenu (što ćemo i prihvatit obnovom polja **d** i **p**) dok bi dolazak u vrh *z* povećao ukupnu cijenu puta od početnog vrha *s* (Slika 8.44 f)). Stoga ovu zadnju alternativu ne prihvaćamo te trenutno i dalje najkraći put od vrha *s* prema vrhu *z* ide preko vrha *t*, i ne uključujući vrh *x*, uz cijenu 4. U novoj iteraciji uzimamo iz topološkog redoslijeda vrh *y* iz kojeg usmjerenost bridova dozvoljava samo odlazak u vrh *z*. Težina brida w=-2 doprinijet će smanjenju trenutne cijene ukupnog puta od vrha *s* do vrha *z* s 4 na 3 te obnavljamo polja **d** i **p** (Slika 8.44 g)). Konačno posljednji vrh topološkog redoslijeda, vrh *z*, nema bridova s kojima može doseći neki od ostalih vrhova te algoritam nalaženje najkraćeg puta od vrha *s* prema svim ostalim vrhovima završava (Slika 8.44 h)).



Slika 8.44 Primjer nalaženja najkraćeg puta na grafu od odabranog početnog vrha *s* do svih ostalih vrhova grafa, korištenjem topološkog sortiranja. Izvor [17].

9 Mreže računala i komunikacijske mreže

Ovaj tekst je najprije iz perspektive sklopovlja (engl. *hardware*) objasnio što je to računalo i kako je nastalo. Zatim je iz perspektive programske podrške (engl. *software*) objasnio kako pisati programe koji omogućuju računalima izvršavanje određenih zadataka. Danas, međutim, živimo u umreženom svijetu (engl. *networked society*) gdje predviđanja govore da će do 2022. godine preko 4 milijardi ljudi i 18 milijardi strojeva međusobno komunicirati preko jedne globalne mreže računala – Interneta ([18], [19]). U ovom poglavlju ćemo stoga objasniti na koji način se računala spajaju u mreže računala. Ovo poglavlje daje samo osnovni uvod u umrežavanje računala, a više teoretskih detalja je moguće pronaći u knjigama na hrvatskom jeziku [20] (ovo poglavlje tematski i sadržajno odgovara uvodnom dijelu navedene knjige) i [21] te knjigama na engleskom jeziku [22] i [23]. Više praktičnih detalja o umrežavanju računala moguće je pronaći u knjizi na hrvatskom jeziku [24].



Slika 9.1 Različiti tipovi računala, na kojima se izvršavaju raznovrsni programi, spajaju se u globalnu mrežu računala koja se naziva Internet

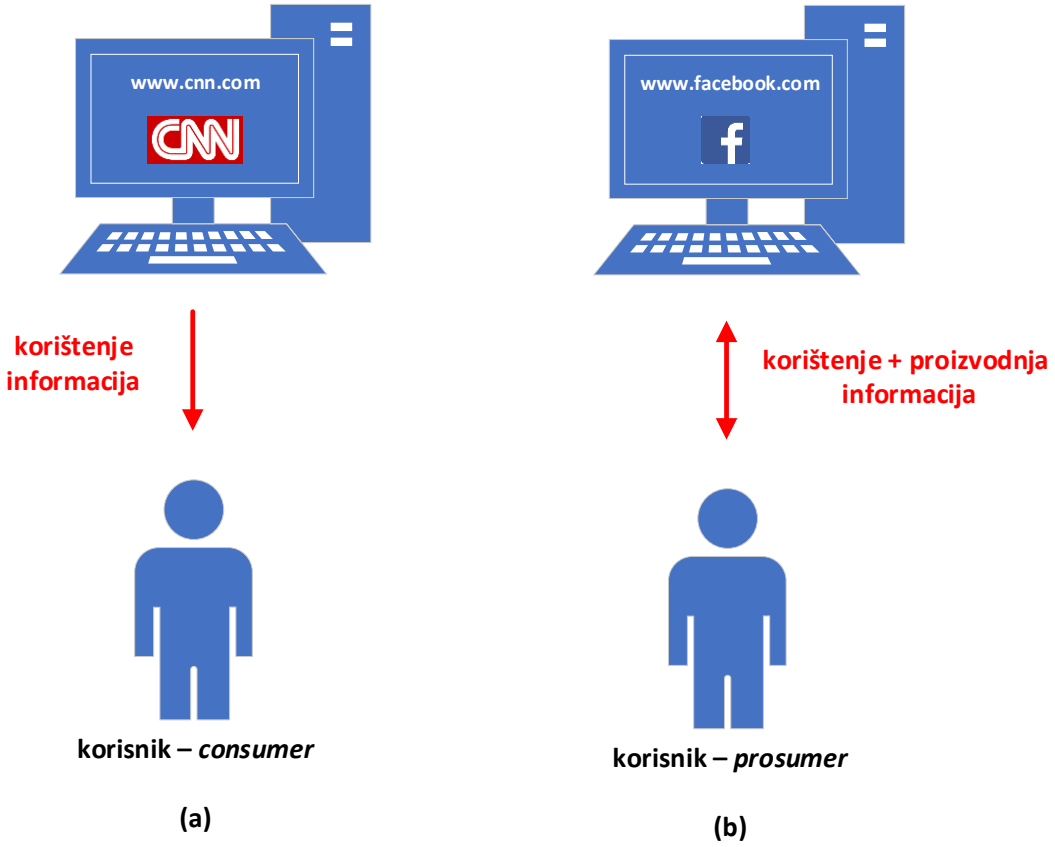
Na slici (Slika 9.1) prikazani su različiti tipovi računala, na kojima se izvršavaju raznovrsni programi, kako se spajaju u globalnu mrežu računala Internet. S lijeve strane slike su korisnički uređaji koji omogućuju ljudima spajanje na Internet (*osobno računalo*, *laptop*, *pametni telefon*), a s desne strane se nalaze umreženi strojevi (engl. *networked machines*) poput umreženog automobila ili umreženog hladnjaka. Takvi strojevi danas više nisu uređaji koje možemo susresti samo u znanstveno-fantastičnim filmovima, već su utkani u našu svakodnevnicu. Na cestama se voze automobili koji u sebi imaju mnogo malih računala te se na tim računalima izvršavaju programi koji se redovito spajaju na Internet (primjerice, usluga navođenja koja u obzir uzima stvarno-vremenske podatke o gužvama na cesti). U kućama se nalaze hladnjaci koji mogu automatski iz dućana naručiti jaja ako su postojeća potrošena.

9.1 Uvod u mreže računala

Cilj umrežavanja računala jest korištenje i/ili pružanje informacijskih i/ili komunikacijskih usluga. Informacijske usluge su one usluge u kojima vrijednost proizlazi iz informacije (primjerice, usluga *World Wide Weba*, skr. *WWW* gdje vrijednost za korisnika proizlazi iz informacije koju je zatražio putem svog web-preglednika). Komunikacijske usluge su one usluge u kojima vrijednost proizlazi iz komunikacije između dva ili više korisnika (primjerice, usluga *Skype* gdje vrijednost za korisnike proizlazi iz činjenice da za nikakav dodatan trošak osim troška korištenja Interneta korisnici mogu međusobno razgovarati bez obzira u kojoj se državi svijeta nalazili; ne samo da mogu razgovarati, već mogu organizirati i video-razgovor između korisnika koji se nalazi u Hrvatskoj i korisnika koji se nalazi u Australiji).

Korisnik (engl. *user*) može biti čovjek ili stroj (engl. *machine*). Korisnik upotrebljava računalnu mrežu za korištenje i/ili pružanje informacijskih i/ili komunikacijskih usluga. Danas su vrlo česti korisnici koji su istovremeno i proizvođači (engl. *producer*) i korisnici informacija (engl. *consumer*), te takve korisnike nazivamo *prosumerima*:

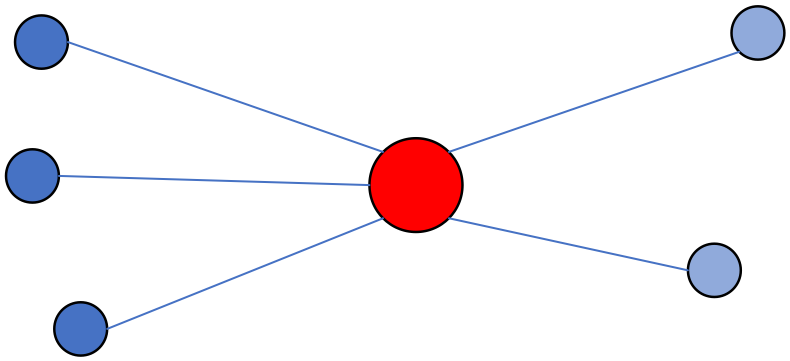
PROducer + conSUMER = PROSUMER



Slika 9.2 Internetski korisnik koji je: (a) *consumer* i (b) *prosumer*

Primjerice, korisnici društvene mreže Facebook su *prosumeri* zato jer istovremeno čitaju obavijesti te gledaju slike i videe koje su objavili njihovi „Facebook prijatelji“, ali i sami objavljuju obavijesti, slike i videe koje onda „konzumiraju“ ostali Facebook korisnici (Slika 9.2). Međutim, isti taj korisnik može biti i „samo“ consumer ukoliko putem svog web-preglednika pristupa nekom portalu s vijestima (engl. *news portal*), poput web-stranice globalne *news* kompanije CNN. Umreženi strojevi također mogu biti *prosumeri*, primjerice umrežena vozila istovremeno koriste stvarno-vremensku uslugu navođenja koja u obzir uzima bogate kontekstne podatke, ali i sami dopridonose kvaliteti usluge navođenja konstantno šaljući podatke o svom kretanju te na taj način obogaćujući podatke o gužvama na cestama.

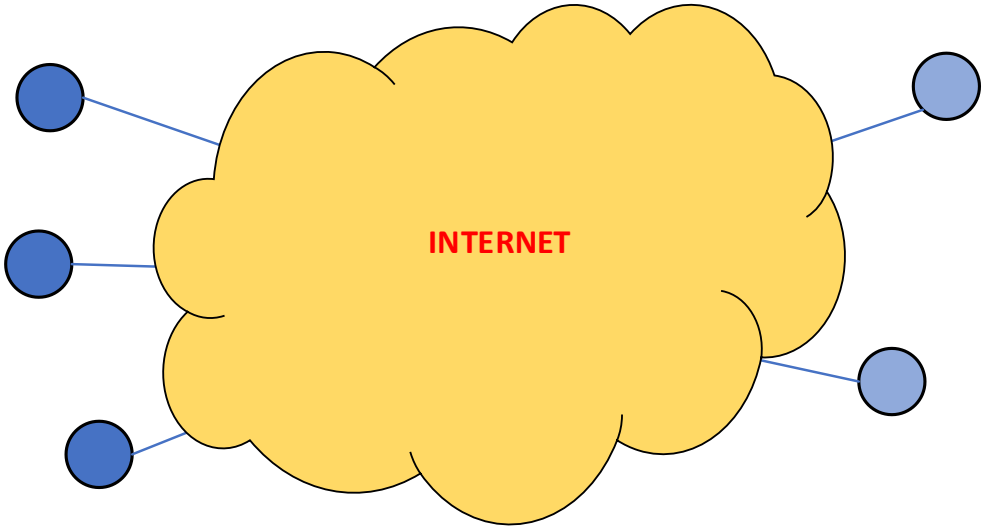
Mrežu računala čine međusobno povezani računalni sustavi na koje se spaja korisnička oprema (komunikacijska, računalna) i druga oprema potrebna za pružanje informacijskih i komunikacijskih usluga. Računalna mreža se može predočiti grafom čiji čvorovi odgovaraju sustavima (komunikacijskim, računalnim), a grane prijenosnim medijima koji ih međusobno povezuju. Na slici (Slika 9.3) možemo vidjeti mrežu računala sa slike (Slika 9.1) prikazanu grafom.



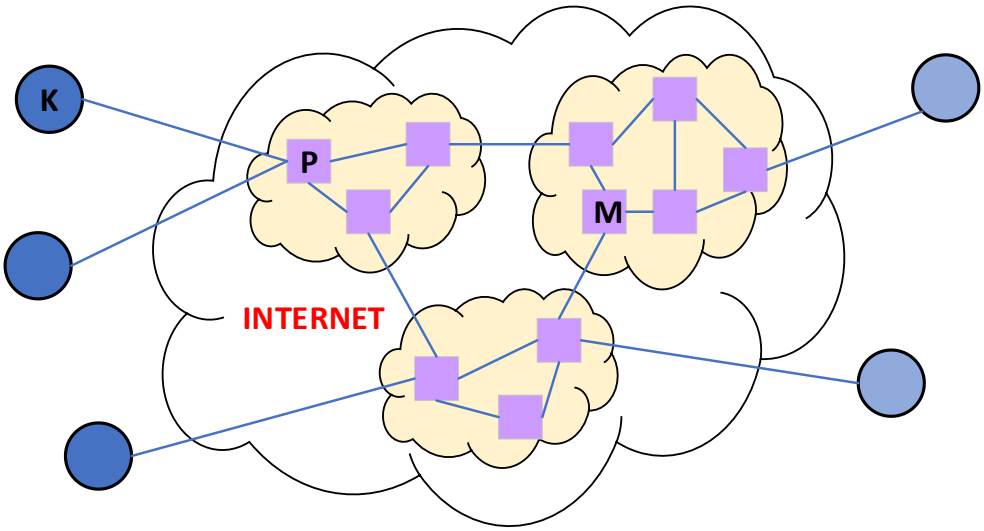
Slika 9.3 Računalna mreža sa slike (Slika 9.1) predočena grafom

U takvom se prikazu razlikuju *krajnji čvorovi* (K) koji predočuju korisničku opremu te međusobno povezani *mrežni čvorovi* (M) koji predočuju komunikacijske sustave. Mrežni čvorovi na koje su priključeni ili s kojima su povezani krajnji čvorovi nazivaju se *pristupnim čvorovima* (P). Mrežni čvorovi provode razne operacije s informacijskim tokovima, poput usmjeravanja kroz mrežu (engl. *routing*), prospajanja, odnosno komutiranja (engl. *switching*) sa svojih ulaza na svoje izlaze, a po potrebi obrađuju i pohranjuju informacije. Potrebni kapacitet mrežnih čvorova ovisi o broju korisnika te namjeni i veličini mreže.

Krajnji korisnici Interneta imaju *logički pogled* na Internet, što znači da vide Internet kao jednu globalnu mrežu. Istina je nešto drugačija – Internet je „mreža mreža“, što znači da je sastavljen od velikog broja podmreža koje međusobno komuniciraju. Ovo poglavlje ima za cilj čitateljima dati „pogled ispod haube“ (*fizički pogled*) i objasniti osnovna pravila koja omogućuju Internetu da funkcionira. Na slici (Slika 9.4) prikazani su logički (Slika 9.4.a) i fizički (Slika 9.4.b) pogled na mrežu računala sa slike (Slika 9.1), gdje je posebno označen jedan krajnji čvor (K), mrežni čvor (M) te pristupni čvor (P).



a) INTERNET - logički pogled: jedna mreža



b) INTERNET - fizički pogled: mreža mreža

Slika 9.4 Logički i fizički pogled na Internet

Sve današnje komunikacijske mreže u svojim krajnjim i mrežnim čvorovima sadrže računala ili procesore, tako da nema većih koncepcijskih razlika između komunikacijske mreže (engl. *communication network*), ili, u užem smislu mreže računala (engl. *computer network*). Stoga ćemo u nastavku poglavlja koristiti terminologiju „mreže računala“, ali isti koncešti vrijede i za „komuniukacijske mreže“.

9.2 Vrste računalnih mreža

Računalne mreže se mogu razvrstati prema različitim kriterijima, kao što su sljedeći:

- *Topologija* (međusobna povezanost računala u mreži);
- *Rasprostranjenost* (područje na kojem se računalna mreža prostire);
- *Vrsta informacije* koja se prenosi računalnom mrežom (podatak, slika, video i/ili drugi);
- *Način razmjene informacije* (komunikacijski kanal ili informacijski paket);
- *Pokretljivost korisnika* (omogućena ili ne); te
- *Namjena* (računalna mreža može biti javna ili privatna).

U nastavku će biti dan opis različitih računalnih mreža s obziorm na gore navedene kriterije.

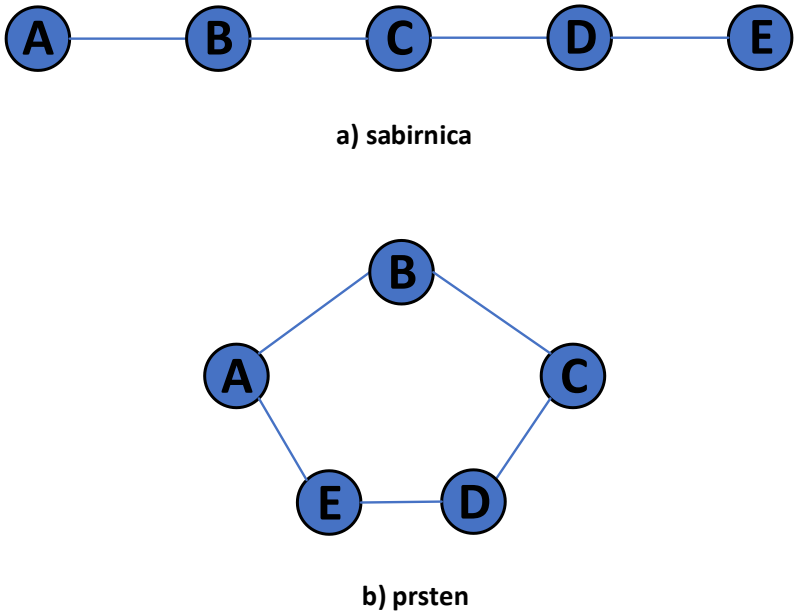
9.2.1 Topologija računalnih mreža

Topologija mreže opisuje međusobnu povezanost računala, pri čemu se razlikuju *fizička* i *logička* topologija. Fizičku topologiju mreže određuje način povezivanja računala prijenosnim medijima, odnosno fizička povezanost računala. Logičku topologiju određuje logička povezanost računala koja proizlazi iz komunikacijskog protokola koji uređuje komunikacijsku povezanost računala, odnosno tok informacija između njih.

Pravilne ili regularne topologije koje se najčešće susreću u mrežama su sljedeće:

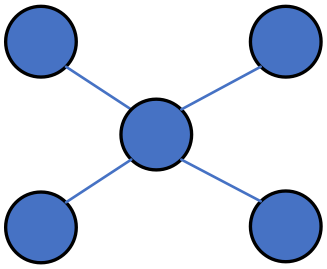
- **sabirnica** (engl. *bus*);
- **prsten** (engl. *ring*);
- **zvijezda** (engl. *star*);
- **stablo** (engl. *tree*); te
- **potpuna povezanost** *(*engl. *fully conected)*.

Fizička i logička topologija mogu se podudarati, ali i razlikovati. Primjerice, računala povezana u fizičku topologiju sabirnice (Slika 9.5.a) mogu ostvarivati komunikaciju logičkom topologijom prstena (Slika 9.5.b), pri čemu se podaci prenose uvijek iz čvora A u čvor B, pa zatim iz čvora B u čvor C i tako redom po principu prstena: A → B → C → D → E → A → ….

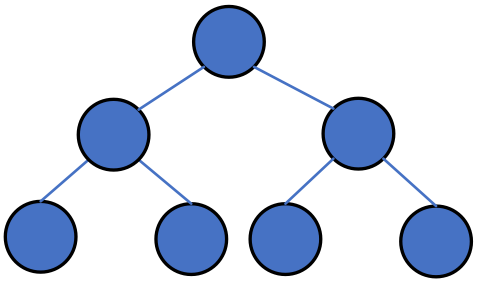


Slika 9.5 Pet čvorova (računala) spojenih u topologiju: a) sabirnice i b) prstena

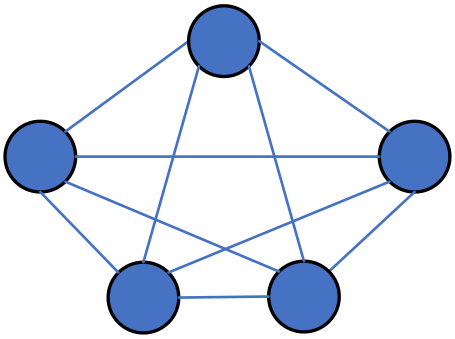
Primjeri računala spojenih u topologije zvijezde i stabla, te primjer potpuno povezanih računala, prikazani su na slici (Slika 9.6).



a) zvijezda



b) stablo



c) potpuna povezanost

Slika 9.6 Čvorovi (računala) spojeni u topologiju: a) zvijezde, (b) stabla i c) potpune povezanosti

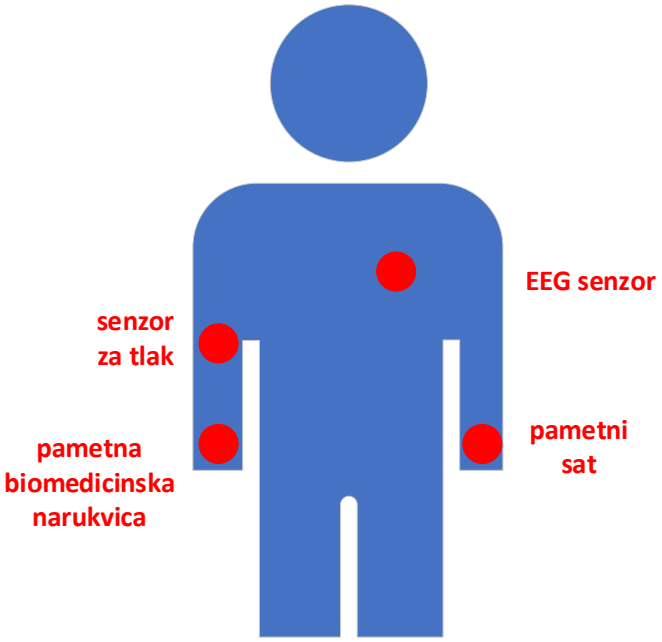
Svaka topologija spajanja računala ima svoje prednosti i nedostatke, te ne postoji generalno „najbolja“ topologija, već se topologija prilagođava specifičnim korisničkim zahtjevima, tehnologijama povezivanja računala te poslovnim politikama pružatelja usluge umrežavanja. Tako je, primjerice, mreža spojena na principu „potpune povezanosti“ najotpornija na kvarove, bilo nekog od umreženih računala koji predstavljaju čvorove u mreži, bilo

prijenosnih medija koji predstavljaju grane u mreži. Ipak, nedostatak takve mrežne topologije je činjenica da broj grana u mreži raste u kvadratnoj ovisnosti prema broju umreženih računala, što znači da je već i za mreže srednje veličine potrebno izgraditi mnogo grana, a svaka grana predstavlja dodatnu investiciju. Možemo zaključiti da su mreže računala spojene na principu „potpune povezanosti“ vrlo robustne s obzirom na kvarove, ali istovremeno vrlo skupe. Slična razmatranja vrijede i za ostale predstavljene topologije, uz načelno pravilo da veći broj grana vodi do veće robusnosti, ali i skuplje izvedbe. Posebno su zanimljivi slučajevi kada imamo „posebne“ čvorove, primjerice u topologiji „zvijezde“, gdje će ispad centralnog čvora (odnosno kvar centralnog računala) uzrokovati potpuni prestanak mrežne povezanosti između bilo koja dva računala u mreži.

9.2.2 Rasprostranjenost računalnih mreža

Iz perspektive *rasprostranjenosti* računalne mreže razlikuju se:

- **mreže širokog područja** (engl. *Wide Area Network*, skr. WAN); Internet je primjer globalne mreže širokog područja koju sačinjavaju različite međusobno povezane mreže, kako je to prikazano na slici (Slika 9.4.b);
- **metropolitanske/gradske mreže** (engl. *Metropolitan Area Network*, skr. MAN); gradske mreže služe za komunikaciju u (većim) urbanim središtima.
- **lokalne mreže** (engl. *Local Area Network*, skr. LAN); lokalnom mrežom ostvaruje se povezivanje računala unutar zgrade ili skupine susjednih zgrada, a spajanjem takve mreže na druge mreže omogućuje se komunikacija u širem području.
- **osobne mreže** (engl. *Personal Area Network*, skr. PAN); osobnu mrežu čine umreženi osobni uređaji istoga vlasnika, poput pametnog telefona, tableta i laptopa koje sve posjeduje ista osoba.
- **tjelesne mreže** (engl. *Body Area Network*, skr. BAN); tjelesna mreža povezuje uređaje smještene na samoj osobi kao što su senzori i biomedicinska oprema, poput pametnog sata, pametne biomedicinske narukvice, umreženog senzora za tlak ili umreženog EEG (engl. *electroencephalogram*) senzora (Slika 9.7).



Slika 9.7 Primjer tjelesne mreže

9.2.3 Vrsta informacija i način razmjene informacija u računalnim mrežama

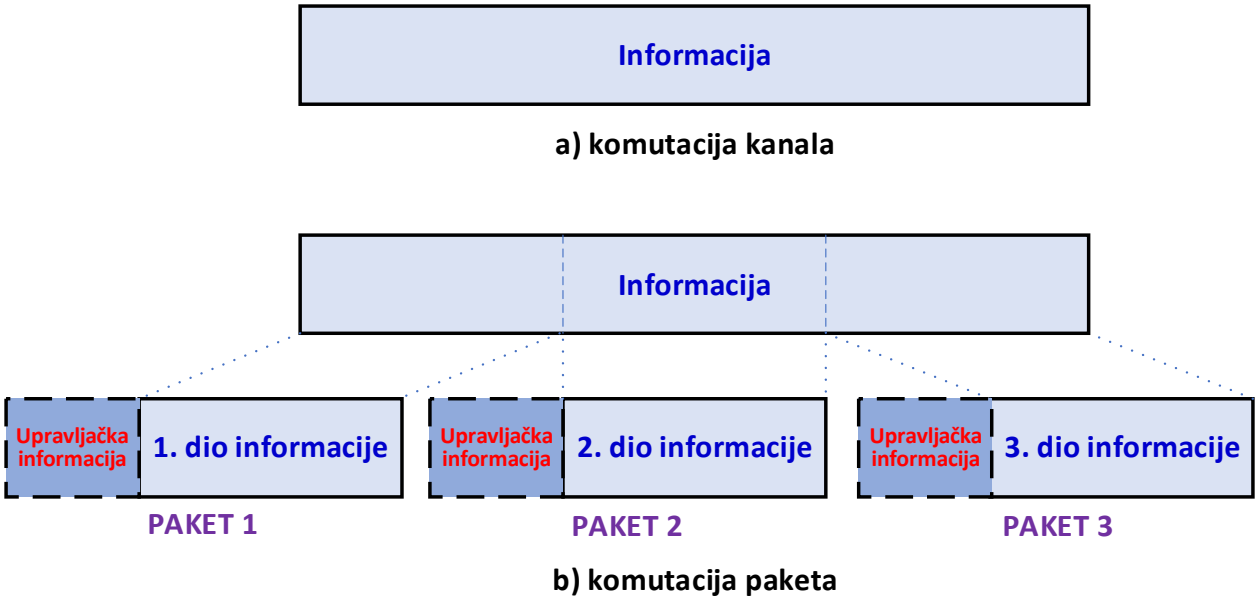
Iz perspektive *vrste informacije*, mreže su se razvijale od onih koje prenose **jednu vrstu informacije** kao što su govor (npr. telefonska mreža) ili video (npr. kabelska televizija), prema mrežama sposobnima za **objedinjeni prijenos više vrsta informacija** (npr. Internet koji objedinjeno prenosi i govor i video).

Iz perspektive *načina razmjene informacije* u mreži razlikujemo komunikaciju:

- komunikacijskim kanalom; te
- informacijskim paketom.

Posljedično, razlikujemo mreže koje funkcioniraju na principu:

- **komutacije kanala**; te
- **komutacije paketa** (Slika 9.8).



Slika 9.8 Razmjena informacija kod mreža koje funkcioniraju na principu: a) komutacije kanala i b) komutacije paketa

Komunikacijski kanal označava put kroz mrežu definiranog kapaciteta između izvorišnog i odredišnog krajnjeg računala koji se dodjeljuje na zahtjev (npr. pozivanjem), ili trajno (npr. najmom), a namijenjen je protoku informacija između izvora i odredišta, s tim da se informacija u izvornom obliku (Slika 9.8.a) šalje putem komunikacijskog kanala. Razmjena informacija komunikacijskim kanalom prikladna je za kontinuirani protok informacija u stvarnom vremenu (npr. kod razgovora) kada je važno održati vremenske uvjete komuniciranja kako se ne bi ugrozila razumljivost zbog prevelikog kašnjenja ili promjene kašnjenja. Međutim, ovakav način komuniciranja ima i nedostatak da je neučinkovit, jer se dodijeljeni kapacitet zauzima (i najčešće naplaćuje) neovisno o tome da li se i koliko se informacije stvarno prenosi.

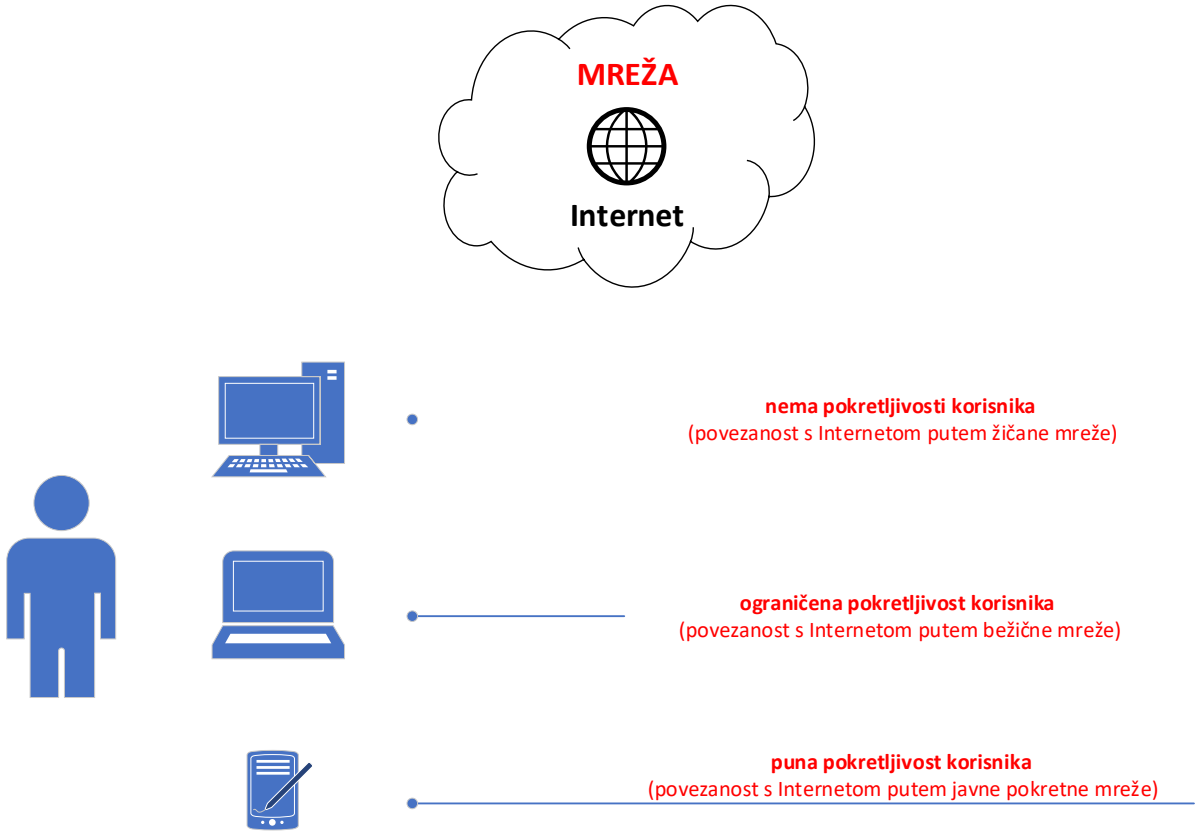
Informacijski paket drugi je temeljni način razmjene informacije. Informacija se dijeli na dijelove kojima se dodaje zaglavlje s upravljačkom informacijom i tako se formiraju paketi (ili datagrami) koji se prenose mrežom (Slika 9.8.b gdje je informacija podijeljena na tri dijela). Zaglavlje uključuje adresu na temelju koje se paketi usmjeravaju od izvora do odredišta. Pri paketskom komuniciranju se mrežni resursi (čvorovi i grane) zauzimaju samo

tijekom prijenosa paketa, a broj i veličina paketa može se prilagoditi količini informacije koja se prenosi. Tako se u razdobljima kad jedan izvor ne odašilje pakete, istim putem mogu prenositi paketi iz drugih izvora. Na taj način se postiže učinkovita uporaba mrežnih resursa, ali je teže održati vremenske uvjete komuniciranja (kašnjenje i promjena kašnjenja), što je nedostatak u odnosu na komutaciju kanala. S obzirom na to da se pri komutaciji paketa mrežni resursi ne rezerviraju unaprijed (prigodom uspostave veze) može doći do kašnjenja paketa koji čekaju na daljnje usmjeravanje u nekom mrežnom čvoru. Uz to, dio se raspoloživog kapaciteta troši na prijenos upravljačke informacije (zaglavlja paketa). S obzirom na takvo postupanje s paketima, može se dogoditi da paketi namijenjeni istom odredištu prolaze različitim putovima i na odredište stignu u redoslijedu koji nije istovjetan onome u kojem su odaslani. Na to utječu kapacitet i trenutno opterećenje putova kojima prolaze, tako da kasnije odaslani paket može proći kraćim putom i brže, te biti na odredištu primljen ranije.

9.2.4 Pokretljivost korisnika i namjena računalnih mreža

Iz perspektive *pokretljivosti korisnika*, računalna mreža može:

- **omogućavati punu pokretljivost korisnika** (npr. javna pokretna mreža četvrte generacije – mreža 4G [25] – koja omogućava da se korisnik vozi automobilom po autocesti i kontinuirano bude spojen na Internet);
- **omogućavati ograničenu pokretljivost korisnika** (npr. bežična mreža WiFi [26] koja omogućava da se korisnik bežično spoji na lokalnu mrežu u nekoj prostoriji te se slobodno kreće unutar navedene prostorije bez gubljenja veze); ili
- **ne omogućavati pokretljivost korisnika** (npr. lokalna mreža ostvarena žičanom tehnologijom (engl. *wired LAN*) koja ne omogućava da se korisnik, zajedno sa svojim umreženim računalom, kreće).



Slika 9.9 Primjeri pokretljivosti korisnika

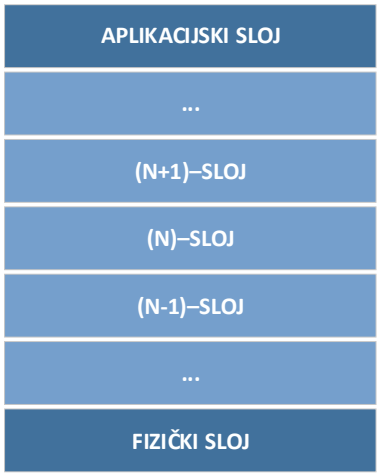
Namjena mreže može biti **javna** ili **privatna**. Javna mreža (engl. *public network*) dostupna je korisnicima kroz ugovorni odnos s mrežnim operatorom (engl. *network operator*), dok je privatna mreža (engl. *private network*) namijenjena ograničenoj skupini korisnika unutar iste zajednice, poput vojne mreže (engl. *military network*) ili korporacijske mreže (engl. *corporate network*).

9.3 Arhitektura mreže

Arhitektura mreže određuje njen izgled i način rada, a može se opisati dijelovima mreže te njenim slojevima. Dijelovi mreže definiraju se njenom horizontalnom dekompozicijom (npr. Slika 9.3), a slojevi vertikalnom dekompozicijom. Kako bi se smanjila složenost oblikovanja mreža, vertikalnom podjelom funkcija uvode se mrežni slojevi (engl. *network layer*), od

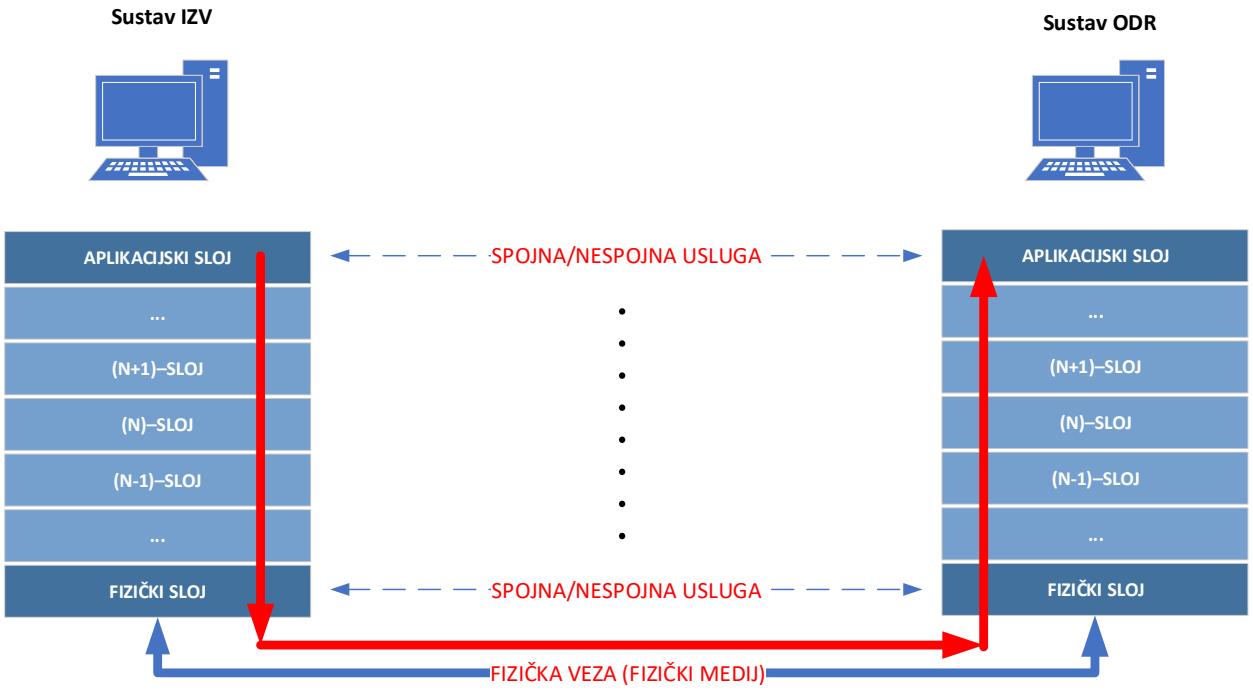
prijenosa informacije fizičkim medijem u najnižem sloju (koji se naziva „fizički sloj“), do usluga i primjena na raspolaganju korisnicima u najvišem sloju (koji se naziva „aplikacijski sloj“) (Slika 9.10). Vertikalnom dekompozicijom na slojeve svakom se sloju dodjeljuju funkcije i specificiraju sučelja sa susjednim slojevima kako bi viši sloj mogao koristiti uslugu nižeg sloja (primjerice, *N-sloj* na slici (Slika 9.10) koristi uslugu nižeg *(N-1)-sloja* te pruža uslugu višem *(N+1)-sloju*). Osnovna je zamisao da svaki sloj nudi usluge višem sloju, prikrivajući rješenje usluga koje omogućuju niži slojevi. Najviši je uvijek aplikacijski sloj koji predstavlja usluge i aplikacije za korisnike, a najniži fizički sloj koji omogućuje stvarni prijenos informacije fizičkim medijem. Slojevita arhitektura računalnih mreža prikladna je i primjenjuje se za sljedeće namjene:

- definiranje koncepata i normiranje računalnih mreža;
- utvrđivanje pravila povezivanja uređaja u računalnu mrežu te mreža međusobno;
- stvaranje otvorenih rješenja, neovisnih o proizvođaču opreme ili mrežnom operatoru.



Slika 9.10 Vertikalna dekompozicija računalne mreže na slojeve

Komunikacija između *(N)-sloja* na izvorišnoj strani i *(N)-sloja* na odredišnoj strani ne odvija se izravno, već posredno. Naime, komunikacija se obavlja unutar izvorišnog sustava, vertikalno od najvišeg sloja prema najnižem na izvorišnoj strani (*sustav IZV*), zatim fizičkim medijem do odredišne strane (*sustav ODR*), pa opet vertikalno, od najnižeg prema najvišem sloju (Slika 9.11).



Slika 9.11 Komunikacija između slojeva na izvorišnoj i odredišnoj strani

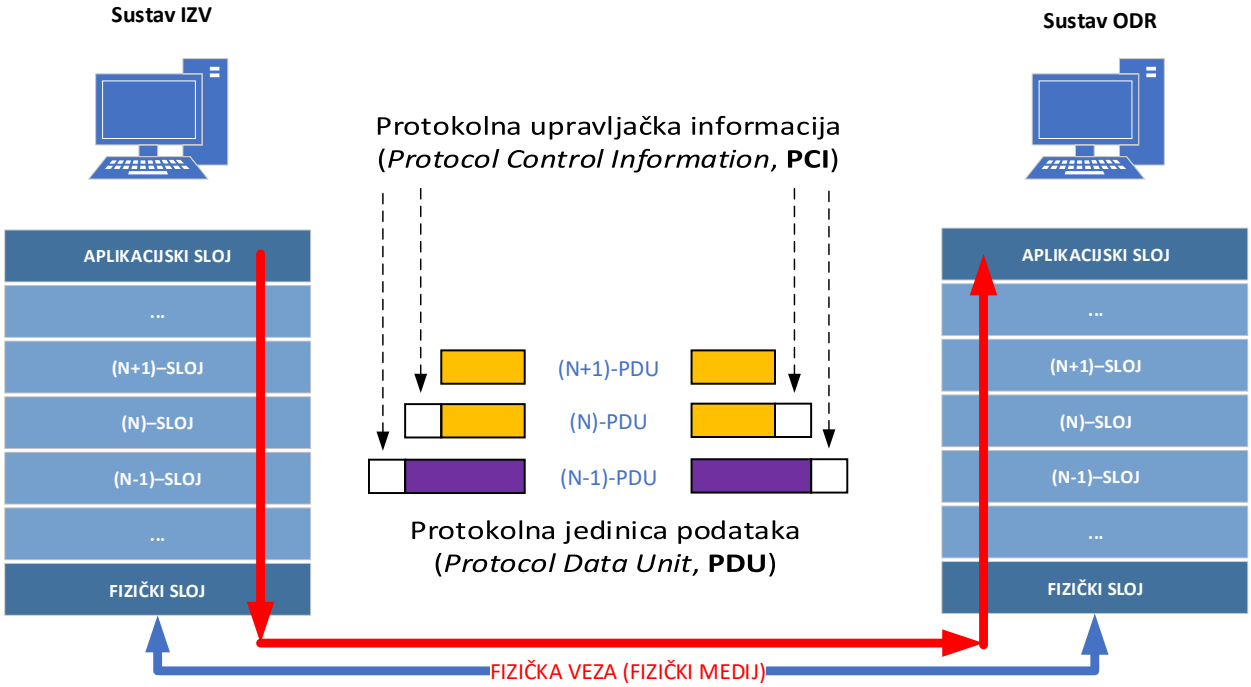
Razmjena jedinica podataka između *(N)-sloja* na izvorišnoj strani i *(N)-sloja* na odredišnoj strani može se provesti:

- **spojnom uslugom** (engl. *connection-oriented service*); ili
- **nespojnom uslugom** (engl. *connectionless service*).

Spojna usluga nalikuje telefonskom pozivu i razgovoru, a nespojna slanju i primanju pisama putem pošte. Kod spojne usluge razmjeni jedinica podataka prethodi uspostavljanje veze, a po završetku razmjene podataka veza se prekida. Sve jedinice podataka razmjenjuju se na isti način koji određuje „veza" (npr. usmjeravaju se istim putem, isporučuju se u redoslijedu u kojem su poslane i dr.), a veza može biti:

- *stvarna*, kada se jedinice podataka prenose istim fizičkim kanalom; ili
- *logička*, kada se posebno obilježavaju jedinice podataka kako bi se označila njihova pripadnost nekoj komunikaciji.

Kod nespojne se usluge svaka jedinica podataka razmjenjuje, odnosno usmjerava i isporučuje na odredištu neovisno o ostalima. Primjer nespojne usluge je datagramski prijenos paketa u Internetu. Datagrami se prenose od izvora do odredišta tako da se prosljeđuju i usmjeravaju između čvorova svaki za sebe, pa se ne može jamčiti njihov redoslijed na odredištu. Ujedno nema mehanizama kojima bi se jamčila isporuka i potvrdilo primitak datagrama na odredištu, pa takva komunikacija nije pouzdana. Kod spojne se usluge svaka jedinica podataka razmjenjuje, odnosno usmjerava i isporučuje na odredištu u međuovisnosti o ostalim jedinicama podataka koje pripadaju istoj komunikaciji. Primjer spojne usluge je prijenos informacija u javnoj telefonskoj mreži (engl. *Public Telephone Switched Network*, skr. PTSN). U slučaju mreže PTSN informacije se prenose od izvora do odredišta tako da se prosljeđuju i usmjeravaju između čvorova svaka istim putem komunikacijskim kanalom između izvorišnog i odredišnog krajnjeg računala koji se dodijeljen na zahtjev, također čuvajući redoslijed informacija s obzirom kako nastaju na izvoru.



Slika 9.12 Protokolna jedinica podataka (PDU) i protokolna upravljačka informacija (PCI)

Protokolna jedinica podataka (engl. *Protocol Data Unit*, skr. PDU) naziv je za strukturiranu informaciju koja se prenosi unutar *(N)-sloja*, između izvora i odredišta (Slika 9.12). Svaki sloj stvara svoju protokolnu jedinicu podataka koju predaje nižem sloju. Primjerice, *(N+1)-sloj* predaje *(N+1)-PDU* nižem, *(N)-sloju*. *(N+1)-PDU* predstavlja uslužnu jedinicu podataka (engl. *Service Data Unit*, skr. SDU) za *(N)-sloj* koju on treba poslužiti, odnosno *(N)-SDU*. Nadalje, *(N)-sloj* dodaje protokolnu upravljačku informaciju (engl. *Protocol Control Information*, skr. PCI), odnosno *(N)-PCI*, uslužnoj jedinici podataka *(N)-SDU*. Primjeri protokolne upravljačke informacije su: adresa, oznaka veličine jedinice podataka, zaštitni kod i drugo. Na odredišnoj strani provodi se obrnuti postupak: sloj preuzima protokolnu jedinicu podataka od nižeg sloja, obrađuje i izuzima vlastitu protokolnu upravljačku informaciju i tako sve do najvišeg sloja na kojem se izvodi aplikacija. Primjerice, *(N-1)-sloj* obrađuje i izuzima protokolnu upravljačku informaciju *(N-1)-PCI* sadržanu u *(N-1)-PDU* tako da *(N)-sloj* preuzima *(N-1)-SDU* kao svoju protokolnu jedinicu podataka, odnosno *(N)-PDU*. Jedinice podataka između slojeva prenose se transparentno, jer *(N-1)-sloj* ne zadire u podatke koje treba proslijediti višem sloju, odnosno *(N)-PDU*. Svaki sloj obavlja postavljene funkcije na temelju protokolne upravljačke informacije.

9.4 Komunikacijski protokol

Komunikacijski protokol obuhvaća skup pravila, formata i postupaka za razmjenu informacije kojim se uređuje međusobno djelovanje umreženih računala. Specifikacija svakog komunikacijskog protokola obuhvaća sljedeće elemente:

- usluga koju protokol pruža;
- rječnik protokolnih jedinica podataka;
- format protokolnih jedinica podataka; i
- pravila razmjene protokolnih jedinica podataka.

Uslugu protokola opisuju njegova namjena te način rada i rukovanje pogreškama. Namjenu protokola određuju njegova funkcija u okviru protokolnog sloja kojem pripada, suradnja s drugim protokolima u istom, višem ili nižem sloju te mreža u kojoj se primjenjuje. Način rada može obuhvatiti jednosmjernu (engl. *simplex*), dvosmjernu (engl. *duplex*) ili polu-

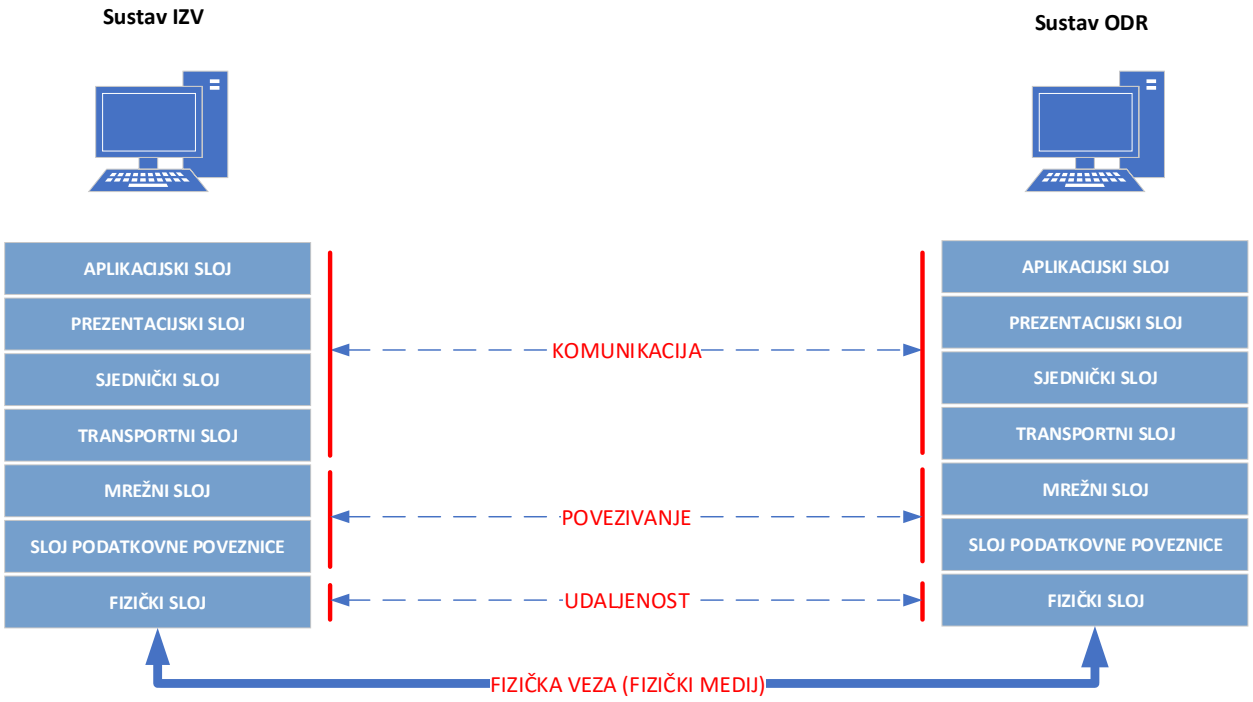
dvosmjernu (engl. *half-duplex*), odnosno naizmjeničnu razmjenu jedinica podataka spojnom ili nespojnom uslugom od točke do točke, ili od jedne prema više točaka. Upravljanje pogreškama može biti riješeno njihovim otkrivanjem ili ispravljanjem, ili prepušteno drugim protokolima s kojima razmatrani protokol surađuje.

Komunikacijski protokoli se u praksi razvrstavaju prema sloju kojem pripadaju (npr. aplikacijski), po dijelu mreže u kojem se primjenjuju (npr. lokalna mreža), vrsti informacije (npr. korisnička informacija ili upravljačka informacija) i drugome. Komunikacijskih protokola ima mnogo, a procesi konvergencije u mrežama dovode do smanjenja broja komunikacijskih protokola u uporabi, kao i razlika među njima.

Skup protokola koji se primjenjuju u slojevima nekog modela ili sustava naziva se protokolni složaj (engl. *protocol stack*).

9.5 Internetski protokolni složaj

Osnovni slojeviti modeli koji definiraju funkcioniranje globalne mreže Internet su referentni model povezivanja otvorenih sustava (engl. *Open System Interconnection Reference Model*, skr. OSI RM) [27] te referentni model TCP/IP, odnosno internetski model (engl. *TCP/IP Reference Model*) [28]. Referentni model OSI RM, na koji će se fokusirati nastavak poglavlja, sadrži sedam slojeva (Slika 9.13) čije se uloge slojeva međusobno nadopunjuju. Osnovna zadaća fizičkog sloja, kao najnižega, je svladavanje udaljenosti pri komuniciranju. Sloj podatkovne poveznice i mrežni sloj za zadaću imaju povezivanje entiteta koji komuniciraju, dok je zadaća transportnog i viših slojeva sama komunikacija kojom se ostvaruju aplikacije i usluge. U nastavku su detaljnije opisane funkcionalnosti svakog od sedam slojeva modela OSI RM.



Slika 9.13 Referentni model *Open System Interconnection Reference Model* (OSI RM)

9.5.1 OSI RM sloj 1: Fizički sloj i prijenosni medij

Fizički sloj (engl. *physical layer*) obavlja funkcije potrebne za ostvarivanje komunikacije fizičkim prijenosnim medijem između dva ili više računala. Jedinica podataka za fizički sloj je **bit**, a osnovne funkcije uključuju:

- prijenos slijeda bita; te
- te mehaničko, električko, odnosno fotoničko te vremensko sučelje s prijenosnim medijem.

Drugim riječima, prijenosni medij se mehanički priključuje na računalni sustav ili uređaj, pri čemu se provodi prilagodba električkog ili optičkog signala prijenosnom mediju, uz vremensko usklađivanje prijenosa signala između predajnika na jednom i prijamnika na drugom komunicirajućem sustavu.

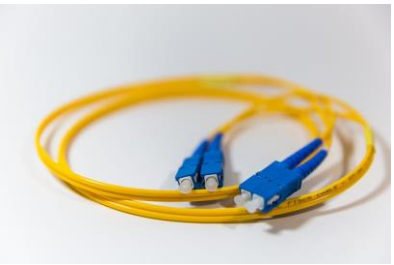
Razlikujemo dvije osnovne vrste prijenosnog medija:

- omeđeni; te
- neomeđeni.

Najčešće korišteni omeđeni prijenosni mediji, koji omogućuju žičani (engl. *wired*) prijenos, su (Slika 9.14):

- *(bakrena) parica* (engl. *Unshielded Twisted Pair*, UTP);
- *optičko vlakno* (engl. *fiber optic cable*);
- *koaksijalni kabel* (engl. *coaxial cable*) koji je bio puno zastupljeniji u ranijim fazama umrežavanja računala krajem 20. stoljeća, dok se danas vrlo rijetko koristi za umrežavanje računala.

(a) bakrena parica



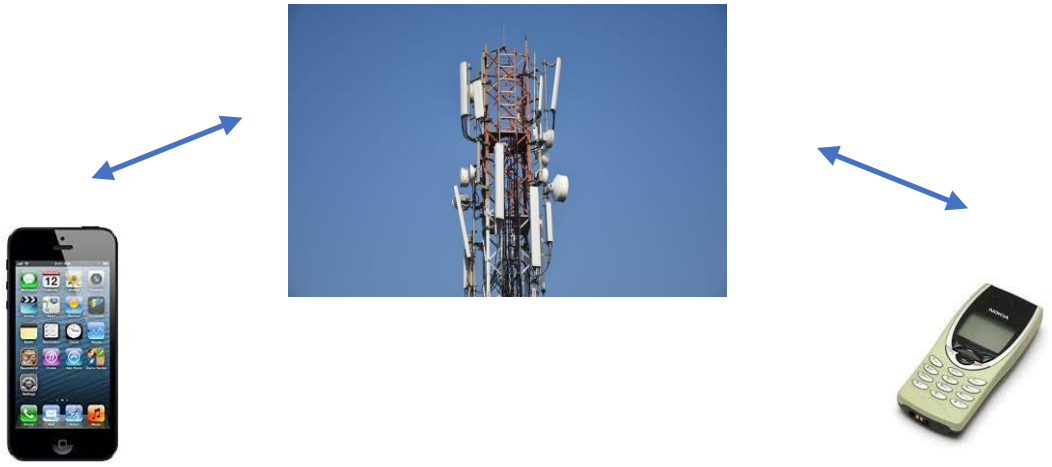
(b) optičko vlakno

(c) koaksijalni kabel

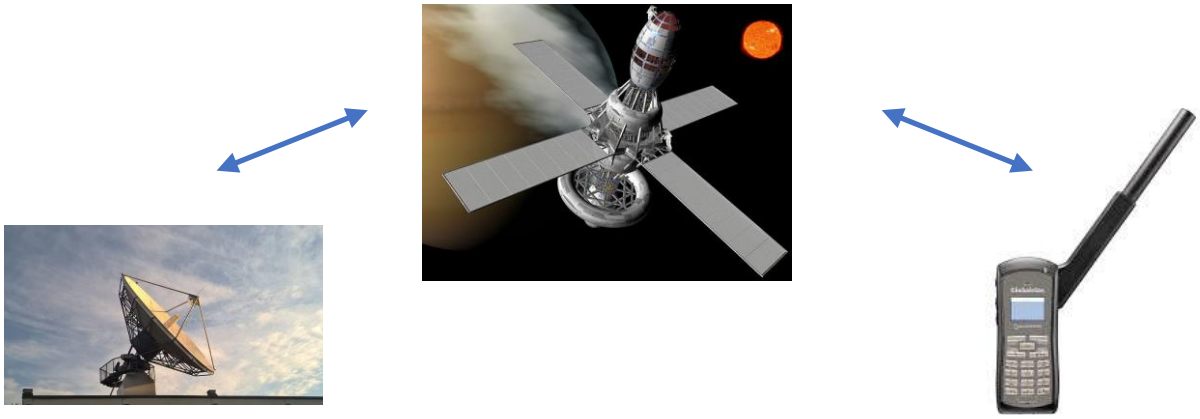
Slika 9.14 Primjeri omeđenih prijenosnih medija

Najčešće korišteni neomeđeni prijenosni medij, koji omogućuje bežični (engl. *wireless*) prijenos, je (Slika 9.15):

- *radijski prijenos*; dok su za neke specifične uporabe mogući i
- *infracrveni prijenos*;
- *optički (laserski) prijenos*;
- *satelitski prijenos*; te
- *mikrovalni prijenos*.



(a) radijski prijenos



(b) satelitski prijenos

Slika 9.15 Primjeri neomeđenih prijenosnih medija

Ako usporedimo funkcioniranje mreže računala s prometnom mrežom gdje vozila putuju od izvora do odredišta (Slika 9.16), tada možemo reći da fizički sloj određuju tehnologiju fizičkog puta koja može biti, primjerice, *cesta* (za automobile) ili *pruga* (za vlakove). Unutar iste tehnologije fizičkog puta također imamo podjele na podtipove, poput podjele cesta na lokalne ceste koje su sporije i povezuju mjesta koja nisu jako udaljena te autoceste koje su brže i povezuju udaljenija mjesta. Pruge pak možemo podijeliti na pruge s tračnicama kojima voze dizelski ili električni vlakovi te *Hyperloop* pruge gdje brzi vlakovi koji rade na principu magnetske levitacije putuju kroz velike vakuumske cijevi.

(a)

(b)

Slika 9.16 Analogija mreže računala s prometnom mrežom: fizički sloj određuje tehnologiju fizičkog puta koja može biti (a) cesta ili (b) pruga

9.5.2 OSI RM sloj 2: Sloj podatkovne poveznice i lokalna mreža

Sloj podatkovne poveznice (engl. *data link layer*) omogućuje **povezivanje dva izravno povezana** (susjedna) računala. Jedinica podataka za sloj podatkovne poveznice je ograničeni niz bita koji se naziva **okvir** (engl. *frame*). Osnovne funkcije sloja podatkovne poveznice su prijenos okvira od točke do točke (engl. *point-to-point*) ili od jedne točke do više točaka (engl. *point-to-multipoint*).

Zadaća sloja podatkovne poveznice je omogućiti izravno povezivanje dvaju susjednih računala ili jednog računala s više susjednih računala i prijenos informacija između njih. Sloj podatkovne poveznice smješten je iznad fizičkog sloja koji mu pruža uslugu prijenosa slijeda bita fizičkim prijenosnim medijem. Povezivanjem, od točke do točke ili od jedne točke do više točaka, u sloju podatkovne poveznice ostvaruju se pretpostavke za pružanje usluge mrežnom sloju, odnosno povezivanje u mreži.

Sloj podatkovne poveznice omogućuje povezivanje računala u lokalnu mrežu (LAN). Najpoznatiji i najrašireniji protokol za lokalnu mrežu *Ethernet* definirao je i razradio industrijski konzorcij u kojem su sudjelovali proizvođači računalne opreme Digital Equipment Corporation, Intel i Xerox (DIX). Objavljene su dvije norme, Ethernet I (1980. g.) i Ethernet II (1982. g.) kojima je specificirana lokalna mreža sa sabirničkom fizičkom topologijom i slučajnim pristupom mediju. Rad konzorcija DIX nastavio je odbor IEEE

802.3 koji razvija istoimenu normu za lokalne mreže. Norme Ethernet i IEEE 802.3 nisu u potpunosti jednake, ali je za tu vrstu mreža uobičajen zajednički naziv Ethernet.

Ako usporedimo funkcioniranje mreže računala s prometnom mrežom gdje vozila putuju od izvora do odredišta (Slika 9.17), tada možemo reći da sloj podatkovne poveznice omogućuje prometovanje automobila između dvije izravno povezane zgrade koje se nalaze u istoj ulici.

(a)

(b)

Slika 9.17 Analogija mreže računala s prometnom mrežom: sloj podatkovne poveznice određuje omogućuje prometovanje između dvije izravno povezane zgrade (b) koje se nalaze u istoj ulici (a)

9.5.3 OSI RM sloj 3: Mrežni sloj

Mrežni sloj (engl. *network layer*) središnji je sloj protokolnog složaja. To je sloj po kojem se mreža „prepoznaje", a koji omogućuje povezivanje između krajnjih računala, izravno ili preko među-čvorova. Za razliku od sloja podatkovne poveznice koji rješava povezivanje od točke do točke, mrežni sloj zadužen je za **povezivanje s kraja na kraj mreže** (engl. *end-to-end*), putem više međusobno povezanih točaka. Izvedbeno, mrežna povezanost ostvaruje se nizom podatkovnih poveznica na putu između izvora i odredišta kao krajnjih računala, tj. od izvora do prvog među-čvora, od njega do sljedećeg među-čvora i tako redom, sve do poveznice zadnjeg među-čvora i odredišta.

Jedinica podataka mrežnog sloja ovisna je o vrsti mreže, npr. to je **paket** u paketskim mrežama, a osnovne funkcije ovog sloja obuhvaćaju usmjeravanje jedinica podataka i

međusobno povezivanje mreža i podmreža, uz mogućnost upravljanja pogreškama i upravljanja tokom.

Mrežni sloj treći je sloj u referentnom modelu OSI, smješten između transportnog sloja i sloja podatkovne poveznice. Najpoznatiji i najrašireniji protokol koji definira funkcionalnosti mrežnog sloja je protokol IP (engl. *Internet Protocol*).

Ako usporedimo funkcioniranje mreže računala s prometnom mrežom gdje vozila putuju od izvora do odredišta (Slika 9.18), tada možemo reći da mrežni sloj omogućuje prometovanje automobila između dvije zgrade koje se mogu nalaziti u različitim državama, što znači da će automobil na putu proći kroz niz raskršća. Upravo mrežni sloj u Internetu je taj koji će definirati treba li automobil na raskršću skrenuti desno ili lijevo, ili pak nastaviti ravno.

(a)

(b)

Slika 9.18 Analogija mreže računala s prometnom mrežom: mrežni sloj omogućuje prometovanje između različitih država (a) prolaskom kroz niz raskrižja (b)

9.5.4 OSI RM sloj 4: Transportni sloj

Transportni sloj (engl. *transport layer*) zadužen je za **komunikaciju s kraja na kraj mreže**, uz zadovoljenje kriterija transparentnosti. Semantička transparentnost odgovara razmjeni podataka bez pogrešaka, a vremenska transparentnost razmjeni podataka uz najmanje moguće kašnjenje. Pogreške su uvijek moguće, tako da se semantičku transparentnost ne može ostvariti bez otkrivanja i ispravljanja pogrešaka, odnosno bez upravljanja pogreškama. Isto tako, preveliki broj jedinica podataka odaslan u mrežu u kojoj se ne upravlja tokom

mogao bi dovesti do zagušenja mreže, gubitka jedinica podataka i nemogućnosti njihove isporuke na odredištu. Na vremensku transparentnost utječe složenost obrade jedinica podataka na izvoru, svakom čvoru na putu i samom odredištu koja izaziva njihovo kašnjenje, pa se često, kako bi se postigla vremenska transparentnost i jedinica podataka isporučila na vrijeme, namjerno zanemaruje semantička transparentnost.

Različite usluge i aplikacije imaju različite zahtjeve. Tako je za dostup podacima na webu (primjerice usluga *World Wide Web*, skr. *WWW*) važna semantička transparentnost, dok razumljivost govora (primjerice usluga *Skype*) nije moguće postići bez vremenske transparentnosti, odnosno dovoljno malog kašnjenja. Najpoznatiji i najrašireniji protokoli koji definiraju funkcionalnosti transportnog sloja su Internetu su protokoli TCP (engl. *Transmission Control Protocol*) i UDP (engl. *User Datagram Protocol*).

(a)

(b)

Slika 9.19 Analogija mreže računala s prometnom mrežom: transportni sloj osigurava dolazak na odredište, prilagođavajući putovanje stanju na cesti (primjerice, gužva u prometu (a)) i eventualnim problemima tijekom puta (primjerice, prometna nesreća (b))

Ako usporedimo funkcioniranje mreže računala s prometnom mrežom gdje vozila putuju od izvora do odredišta (Slika 9.19), tada možemo reći da transportni sloj osigurava dolazak automobila na udaljeno odredište, što znači da prilagođava putovanje stanju na cesti i osigurava oporavak od eventualnih stresnih situacija. Upravo transportni sloj u Internetu je

taj koji će definirati treba li automobil ići brže ili sporije (upravljanje tokom), te omogućiti popravak automobila ako mu se dogodi nešto loše na putu (upravljanje pogreškama).

9.5.5 OSI RM sloj 5: Sjednički sloj

Sjednički sloj ili sloj sesije (engl. *session layer*) usklađuje rad sustava koji međusobno komuniciraju tako da uspostavlja, održava i prekida dijalog, dodjeljuje im prava za komuniciranje, a može im omogućiti nastavljanje komunikacije u slučaju prekida izazvanog smetnjama ili kvarovima.

Ako usporedimo funkcioniranje mreže računala s prometnom mrežom gdje vozila putuju od izvora do odredišta (Slika 9.20), tada možemo reći da sjednički sloj osigurava kontinuitet putovanja bez obzira na potencijalne prekide koji mogu nastati zbog događaja ili problema na nižim slojevima. Upravo sjednički sloj u Internetu je taj koji će prebaciti putovanje s ceste na prugu, ako više nije moguće dalje putovati cestom.

Slika 9.20 Analogija mreže računala s prometnom mrežom: sjednički sloj osigurava kontinuitet putovanja (primjerice, prebacivanjem putovanja s ceste na prugu)

9.5.6 OSI RM sloj 7: Prezentacijski sloj

Za razliku od nižih slojeva koji se bave razmjenom podataka, prezentacijski, odnosno predodžbeni sloj (engl. *presentation layer*) određuje prikaz (sintaksu) i značenje (semantiku) informacije koja se razmjenjuje, tako da propisuje kodove, formate i strukturu podataka.

Ako usporedimo funkcioniranje mreže računala s prometnom mrežom gdje vozila putuju od izvora do odredišta (Slika 9.21), tada možemo reći da prezentacijski sloj osigurava razumijevanje prometnog sredstva kada ono dođe na odredište. Primjerice, na odredište stignu tri kamiona i svaki dopremi po jednu veliku policu. Treba li te tri police spojiti u jedan element prije stavljanja na zid ili staviti svaku policu posebno na zid (sintaksa), određuje prezentacijski sloj. Nadalje, odredište može očekivati dostavu putem prijevoznog sredstva „jaguar“. Je li „jaguar“ predstavlja jaguara životinju ili Jaguara tip vozila (semantika), određuje prezentacijski sloj. Sukladno opisanome, upravo prezentacijski sloj u Internetu je taj koji će definirati „jezik“ komunikacije, kao i strukturu „poruka“.

(a)

(b)

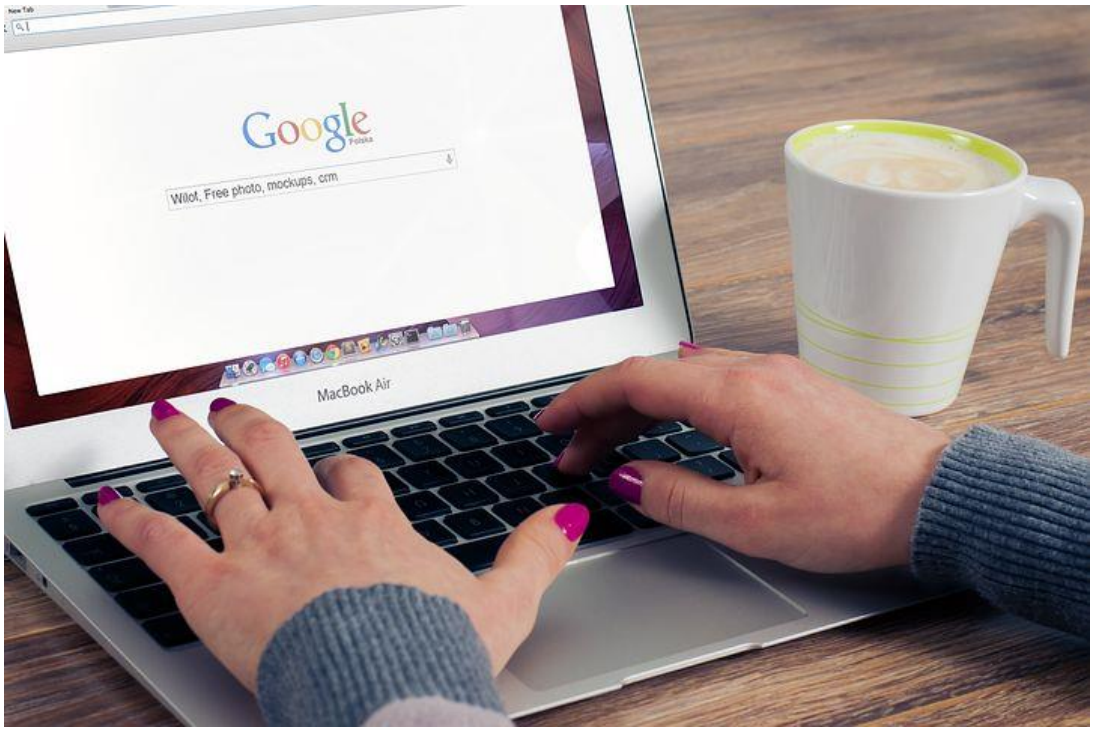
Slika 9.21 Analogija mreže računala s prometnom mrežom: prezentacijski sloj osigurava razumijevanje – je li „jaguar“ predstavlja Jaguara tip vozila (a) ili jaguara životinju (b)

9.5.7 OSI RM sloj 7: Aplikacijski sloj

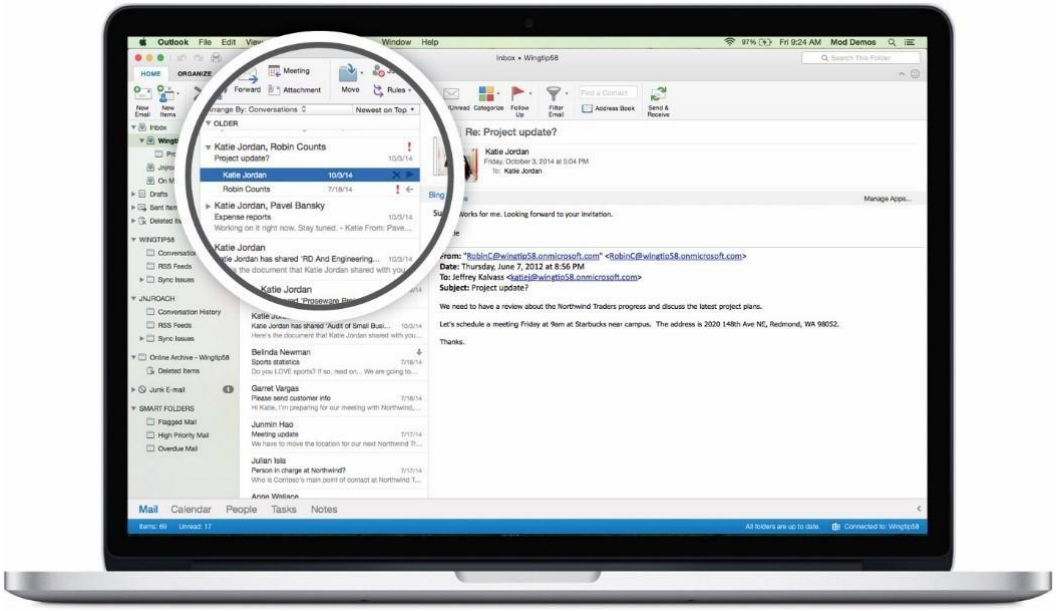
Aplikacijski sloj ili sloj primjene (engl. *application layer*) definira usluge i aplikacije za korisnike, uključujući skup protokola za usluge i aplikacije, a sadrži aplikacijske računalne procese kojima se usluge i aplikacije ostvaruju. Postoje dvije vrste usluga aplikacijskog sloja:

- korisničke usluge; te
- sustavske usluge.

Korisničke usluge su one koje pružaju određenu uslugu krajnjim korisnicima, poput usluge weba (WWW, Slika 9.22) koja je definirana protokolom HTTP (*Hypertext Transfer Protocol*) i usluge elektroničke pošte (engl. *electronic mail*, skr. *e-mail*, Slika 9.23) koja je definirana protokolima SMTP (*Simple-Mail Transfer Protocol*), POP (*Post Office Protocol*) i/ili IMAP (*Internet Mail Access Protocol*).



Slika 9.22 Usluga weba (*World Wide Web*): jedan od najpopularnijih pretraživača web-stranica *Google*



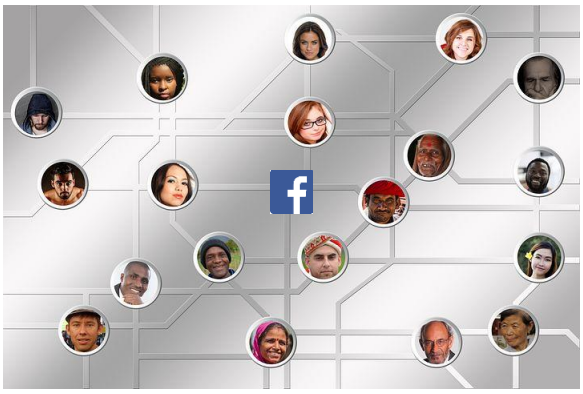
Slika 9.23 Usluga elektroničke pošte (*e-mail*)): jedan od najpopularnijih *e-mail* klijenata *MS Outlook*

Sustavske usluge su one koje pružaju uslugu drugim uslugama aplikacijskog sloja, a ne direktno krajnjim korisnicima te su na taj način ključne za funkcioniranje infrastrukture Interneta. Primjer sustavske usluge je usluga „sustava imenovanja domena“ (engl. *Domain Name System*, skr. DNS) koja je jedna od ključnih internetskih usluga budući da predstavlja „imenik Interneta”. Motivacija za uvođenje DNS-a potiče od potrebe da se razne informacije o mreži, prvenstveno mrežne adrese, povežu s ljudima lakše pamtljivim simboličkim imenima umjesto s numeričkim identifikatorima u izvornom obliku. Na primjer, student koji želi pristupiti web-stranicama Vojnih studija Sveučilišta u Zagrebu ili poslati kolegi poruku elektroničke pošte lako će zapamtiti i primijeniti simboličke adrese, kao „pristupam stranici weba na poslužitelju *„vojni.unizg.hr*”, odnosno, „pišem poruku elektroničke pošte prijatelju na *ime.prezime@vojni.unizg.hr*”. U primjeru pristupanja web-stranici, program web-klijenta (preglednika) je taj koji će „u pozadini”, s pomoću DNS-a, pretvoriti simboličku adresu u numeričku, kako bi mogao pokrenuti vezu prema web-poslužitelju.

Ako usporedimo funkcioniranje mreže računala s prometnom mrežom gdje vozila putuju od izvora do odredišta (Slika 9.24), tada možemo reći da aplikacijski sloj objašnjava razlog i

svrhu putovanja prometnog sredstva, a to može primjerice biti druženje prijatelja (osobe putuju automobilom da bi se našle na odredištu i družile; na sličan način internetska usluga Facebook omogućuje „virtualno“ druženje prijatelja).

(a)



(b)

Slika 9.24 Analogija mreže računala s prometnom mrežom: aplikacijski sloj objašnjava razlog i svrhu putovanja – druženje prijatelja u stvarnom životu (a) i putem internetske usluge *Facebook* (b)

10 Literatura

[1] Turk, S. (1990.). *Arhitektura i organizacija digitalnih računala.* Zagreb: Školska knjiga

[2] Pascal's calculator. (15. 12. 2017.). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/Pascal%27s_calculator

[3] History of computing hardware. (15. 12. 2017.). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/History_of_computing_hardware

[4] Colossus computer. (15. 12. 2017.). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/Colossus_computer

[5] Harwell CADET. (15. 12. 2017.). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/Harwell_CADET

[6] Intel 4004. (15. 12. 2017.). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/Intel_4004

[7] Osborne 1. (15. 12. 2017.). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/Osborne_1

[8] Ribarić, S. (1997.). Naprednije arhitekture mikroprocesora. Zagreb: Element.

[9] U. Peruško, V. G. (2005.). Digitalni sustavi. Zagreb: Školska knjiga.

[10] Boolean algebra. (15. 12. 2017.). Dohvaćeno iz Wikipedia: https://en.wikipedia.org/wiki/Boolean_algebra

[11] L. Euler, Solutio problematis ad geometriam situs pertinentis, Commentarii Academiae Scientiarum Imperialis Petropolitanae, 8 (1736), 128–140

[12] https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg#cite_note-1

[13] M. Guan, *Graphic Programming using odd and even points*, Chinese Mathematics 1, 1962

[14] http://mathworld.wolfram.com/Tree.html

[15] https://en.wikipedia.org/wiki/Prim%27s_algorithm

[16] Kahn, Arthur B. (1962), "Topological sorting of large networks", Communications of the ACM, 5 (11): 558–562.

[17] http://www.cdn.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/

[18] https://www.ericsson.com/en/mobility-report/internet-of-things-forecast

[19] http://www.internetworldstats.com/stats.htm

[20] Lovrek, Ignac; Matijašević, Maja; Ježić, Gordan; Jevtić, Dragan (2017., radna inačica), Komunikacijske mreže, Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, Zagreb

[21] A. Bažant, G. Gledec, Ž. Ilić, G. Ježić, M. Kos, M. Kunštić, I. Lovrek, M. Matijašević, B. Mikac, V. Sinković (2004.), Osnovne arhitekture mreža, Element, Zagreb

[22] Andrew S. Tanenbaum, David J. Wetherall (2010.), Computer Networks, 5/e, Prentice Hall

[23] James F. Kurose, Keith W. Ross (2017.), Computer Networking: A Top-Down Approach Featuring the Internet, 7/e, Pearson

[24] Podobnik, Vedran; Dobrijević, Ognjen; Grgić, Tomislav; Ivešić, Krunoslav (2017., radna inačica), Internetski protokoli u primjeni, Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, Zagreb

[25] http://www.itu.int/pub/R-REP-M.2134-2008/en

[26] http://www.ieee802.org/11/

[27] https://www.webopedia.com/quick_ref/OSI_Layers.asp

[28] http://www.omnisecu.com/tcpip/tcpip-model.php