UDK 004.42
UDK 378
Stručni rad

Dr. sc. Nevzudin Buzađija

OPTIMIZIRANJE KODA U FUNKCIJI RAZVOJA PROGRAMERSKIH VJEŠTINA KOD STUDENATA

Sažetak

Pravilan pristup u pisanju programskog koda i način optimiziranja koda je bitna karakteristika za dobrog programera sa stajališta zahtjeva koji se postavljaju od strane softverske industrije. Da bi se pristupilo pravilnom pisanju koda neophodno je stvoriti temelj kroz formalno i neformalno obrazovanje. U radu će se prikazati pravilan način pisanja koda uz pravilno optimiziranje koda sa ciljem što bržeg izvršavanja aplikacije. Doprinos rada se ogleda u izgradnji vještina kod mladih ljudi koji se školuju za buduće poslove programera. Cilj je da budući programeri percipiraju sliku na koji način mogu da razviju vlastiti stil pisanja koda.

***Ključne riječi**: funkcionalnost, greške, usko grlo, optimizacija koda, straničenje, Paretov princip, interpreteri, kompajleri, C, C#*

Uvod

Prilikom razvoja aplikacija neophodno je da se odabere strategija razvoja, od odabira adekvatnog sistema za upravljanje bazama podataka do programske platforme. Odabir programske platforme za pisanje koda *backend*, koji nije vidljiv korisniku, treba da je usklađen sa problemom koji se rješava. Glavni zadatak *backend* razvoja je osigurati da aplikacija ispravno funkcioniše i da se aplikacija poveže sa korisničkim interfejsom koji je razvijen od *frontend* developera.

Performanse su samo jedan aspekt ukupnog kvaliteta softvera i dobro optimizirani kod je jedan od aspekta ukupnih performansi. Programska arhitektura, detaljni dizajn, upotrebljene strukture podataka i izbor algoritama u najvećem broju slučajeva imaju veći uticaj na performanse programa (vrijeme izvršavanja i zauzeće memorije) nego što to ima efikasnost njegovog programskog koda.

Kvantitativna mjerenja su ključni dio maksimiziranja performansi. Ona su neophodna da bi se pronašli dijelovi programa čije bi optimiziranje performansi zaista dalo rezultate. Većina aplikacija troši veliki dio vremena izvršavanja u malim dijelovima koda. Ne možete znati koji su to dijelovi dok ne izvršite mjerenja. Najbolji način da se pripremite za rad na performansama putem optimiziranja koda (code tuning) tokom inicijalnog programiranja jeste da pišete standardni kod koji je lak za razumijevanje i modifikaciju.

Problemi kod edukacije programera

Najveći problem kod mladih programera što pribjegavaju šabloniziranju prilikom pisanja izvornog koda i što ne razvijaju svoj vlastiti pristup pisanja algoritma za rješavanje nekog problema. Prvenstveno treba da se razvije stil pisanja koda kod pojedinaca i logičko razumijevanje rješavanja nekog problema. Nakon te početne faze nastoji se kod mladih programera izgraditi percepcija da pišu standardni kod koji je lak za razumijevanje i

modifikaciju. Sljedeća faza je da prepoznaju „uska grla“ u kodu i da prepoznaju moguće optimiziranje koda koje će poboljšati algoritam.

Konstruktivizam sugeriše da vježbe programiranja treba odgoditi sve dok rasprava među studentima ne omogući izgradnju dobrog računarskog modela. Prerani pokušaji pisanja programa dovode do grešaka i odgađa razvoj održivih modela [1]. Naime sve više se nastoji kod studenata izazvati uzbuđenje, kreativnost i inovativnost korištenjem metode kreativnog kodiranja [3].

Prerano optimiziranje nije dobar pristup programiranju jer bolje da imate program koji radi, nego da imate brz program koji pada ili daje netačne informacije. Sa druge strane, teško je napraviti uspješnu aplikaciju bez bavljenja optimiziranjem u procesu stvaranja aplikacije. Programer daleko bolje razumije svoju aplikaciju nego kompajler koji nema takve informacije. Podešavanje koda se vrši iz nekoliko razloga:

- zadovoljstvo je uzeti jednu rutinu koja se izvršava za 30 milisekundi, promijeniti par linija koda i smanjiti vrijeme izvršavanja rutine na 5 milisekunde,
- ovladavanje vještinom pisanja efikasnog koda jeste siguran put ka tome da postanete ozbiljan programer.

Pokazat ćemo jedan primjer koda koji je funkcionalan, a koji bi se mogao optimizirati. Ovo je jako bitno prilikom edukacije programera da se razvijaju vještine fazno i da na prvom mjestu bude funkcionalnost, a tek nakon toga u vremenu do implementacije da se isti optimizira.

Programskim kodom je implementiran RC6 algoritam. Algoritam je napravljen prema standardnim parametrima (ključem od 128/192/256 bitova, podatkom od 128 bitova i 20 rundi algoritma, ali je moguće i modificirati ove parametre), a ključ i podatak se unose u heksadecimalnom obliku (u obliku 4 bloka). Program je postavljen da uradi enkripciju, a zatim dekripciju podatka kako bi se provjerila validnost pseudokodova i rezultata.

Za upotrebu inicijalizacije varijable u for petlji potreban je C11/C99.

Ovo je jedan od primjera koji su korišteni:

Ključ: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Podatak: 00000000 00000000 00000000 00000000

Rezultat: 36a5c38f 78f7b156 4edf29c1 1ea44898

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#define w 32
#define r 20
#define P32 0xB7E15163
#define Q32 0x9E3779B9
#define brbajtova (w/8)
#define c ((b+brbajtova-1)/brbajtova)
#define R24 (2*r+4)
#define lgw 5
#define ROTL(x,y) (((x)<<(y&(w-1))) | ((x)>>(w-(y&(w-1)))))
#define ROTR(x,y) (((x)>>(y&(w-1))) | ((x)<<(w-(y&(w-1)))))

unsigned int S[R24-1];

void potkljucevi(unsigned char *K, int b){
  int i,j;
  unsigned int L[(32+brbajtova-1)/brbajtova];
  unsigned int A,B;
  L[c-1]=0;
  for(i=b-1; i>=0; i--)
    L[i/brbajtova]=(L[i/brbajtova]<<8)+K[i];

//implementacija pseudokoda za generisanje kljuceva
S[0]=P32;
  for(i=1; i<=2*r+3; i++)
    S[i]=S[i-1]+Q32;
  A=B=i=j=0;
  int v=R24;
  if (c>v) v=c;
v*=3;
  for(int s=1; s<=v; s++){
    A=S[i]=ROTL(S[i]+A+B, 3);
    B=L[j]=ROTL(L[j]+A+B, A+B);
    i=(i+1)%R24;
    j=(j+1)%c;
  }
```

```
//kraj implementacije pseudokoda za generisanje kljuceva
}

void enkripcija(unsigned int *pt, unsigned int *ct){
   unsigned int t,u,x;
   int i, j;
   unsigned int A=pt[0];
   unsigned int B=pt[1];
   unsigned int C=pt[2];
   unsigned int D=pt[3];

//implementacija pseudokoda za enkripciju
            B+=S[0];
            D+=S[1];
            for (i=2; i<=2*r; i+=2){
               t=ROTL(B*(2*B+1), lgw);
               u=ROTL(D*(2*D+1), lgw);
               A=ROTL(A^t,u)+S[i];
               C=ROTL(C^u,t)+S[i+1];
               x=A;
               A=B;
               B=C;
               C=D;
               D=x;
            }
            A+=S[2*r+2];
            C+=S[2*r+3];

//kraj implementacije pseudokoda za enkripciju
   ct[0]=A;
   ct[1]=B;
   ct[2]=C;
   ct[3]=D;
}

void dekripcija(unsigned int *ct, unsigned int *pt){
            unsigned int t,u,x;
            int i,j;
            unsigned int A=ct[0];
            unsigned int B=ct[1];
            unsigned int C=ct[2];
            unsigned int D=ct[3];

//implementacija pseudokoda za dekripciju
            C-=S[2*r+3];
            A-=S[2*r+2];
            for(i=2*r; i>=2; i-=2){
                        x=D;
                        D=C;
                        C=B;
                        B=A;
                        A=x;
                        u=ROTL(D*(2*D+1), lgw);
                        t=ROTL(B*(2*B+1), lgw);
                        C=ROTR(C-S[i+1], t) ^ u;
                        A=ROTR(A-S[i], u) ^ t;
            }
            D-=S[1];
            B-=S[0];

//kraj implementacije pseudokoda za dekripciju
            pt[0]=A;
            pt[1]=B;
            pt[2]=C;
            pt[3]=D;
}

int main(){
unsigned int blokovi[4];
unsigned int sifrat[4];
   int i;
   printf("Unesite duzinu kljuca:\n\t1. 128-bit\n\t2. 192-
bit\n\t3. 256-bit\n");
   char ulaz;
   while(1){
            ulaz=getch();
            if (ulaz=='1'){i=16; break;}
            else if (ulaz=='2'){i=24; break;}
            else if (ulaz=='3'){i=32; break;}
            }system("CLS");
   unsigned char segkljuc[i];
   printf("Unosi se kljuc od %d bita\n (segmenti su u hex
obliku od 00 do ff):\n", i*8);
   for(int j=0; j<i; j++){
            printf("Segment %d: ", j+1);
            scanf("%02x", segkljuc+j);
            }system("CLS");
printf("(blokovi su u hex obliku 00000000-ffffffff):\n" );
            for(int j=0; j<4; j++){
            printf("Blok %d: ", j+1);
            scanf("%08x", blokovi+j);
            }system("CLS");
   potkljucevi(segkljuc, sizeof(segkljuc));
   enkripcija(blokovi, sifrat);
   printf("Rezultat enkripcije:\n");
   for(i=0; i<4; i++)
            printf("%08x ", sifrat[i]);
            printf("\nRezultat dekripcije:\n");
            dekripcija(sifrat, blokovi);
for(i=0; i<4; i++)
            printf("%08x ", blokovi[i]);
   return 0;
}
```

U programu je implementirana logika za enkripciju podataka, a zatim dekripciju podatka kako bi se provjerila validnost pseudokodova i rezultata. Naravno kod je funkcionalan i daje tačne izlazne rezultate na osnovu unesenih inputa. Međutim, nakon toga je potrebno izvršiti optimiziranje koda u dijelovima pojedinih funkcija. To bi bila druga faza i jako je bitno

u ovakvim logičkim zadacima koji su napisani u C programskom jeziku da budući programeri na osnovu mjerenja uoče „uska grla“ programa i prepoznaju moguća mjesta za optimiziranje koda kao što su funkcije enkripcije i dekripcije podataka. Na ovaj način se može vidjeti koliko su studenti savladali sintaksu i logiku pojedinih dijelova koda. S druge strane, na ovaj način mogu se motivisati studenti da izgrađuju vlastiti stil i pronalaze druge načine rješavanja pojedinih modula unutar postojećeg koda.

Optimiziranje koda kao nadogradnja vještina programera

Optimiziranje programskog koda je praksa modificiranja ispravnog programskog koda u svrhu efikasnijeg izvršavanja istog [8]. Optimiziranje koda su male promjene koje zahvaćaju jednu funkciju ili samo par linija koda, a koje će na globalnom nivou poboljšati efikasnost koda. Kada se izgradi logičko razmišljanje kod budućih programera onda se pristupa fazama koje svakog od njih čini posebnim i koji će izgrađivati svoj stil pisanja algoritama.

Kod optimiziranja koda poznat je **Paretov princip**. Paretov princip je poznat i kao 80/20 pravilo koje kaže da možemo dobiti 80% postignutog rezultata uz 20% optimalno uloženog truda. Programerima teško je pogoditi kojih 20% koda izvršava 80% programa, tako da ako budu optimizirali dok programiraju, potrošit će 80% vremena optimizirajući kod koji ne treba optimizirati. Performanse se uvijek moraju izmjeriti da bi se znalo da su neke promjene pomogle ili odmogle aplikaciji [4]. Rezultati mjerenja se mijenjaju svaki put kada promijenimo okruženje poput korištenih vanjskih dll-ova, verzije frameworka ili prevoditelja, procesora, memorije, i sl. Rezultati koje smo dobili pri jednom mjerenju u jednom okruženju mogu vrlo lako biti drukčiji u drugom okruženju [7].

Danas postoje mnoge zablude po pitanju optimiziranja koda kao što su vrijeme kad se optimizira, šta se optimizira i redoslijed optimiziranja svih dijelova aplikacije. U današnjoj praksi profesionalni programeri pristupaju obazrivo optimiziranju koda i nikada ne idu na štetu funkcionalnosti koda i vremena isporuke koda/aplikacije.

Optimiziranje koda se vrši [6]:

- na globalnom nivou u odnosu na mikro optimiziranje koda,
- identificiranje „uskog grla“ je moguće tek kada se program počne izvoditi u stvarnom okruženju,
- identifikacija modula ili dijela koda koji izvršava veći dio programa,
- optimiziranju se pristupa nakon što je sistem napravljen tako da bi se mogla identifikovati sva „uska grla“,
- izbjegavanje opterećenja sa optimiziranjem koda u toku pisanja koda kako bi se izbjeglo odvlačenje programera od krajnjeg cilja,
- optimiziranje koda, a samim tim ni brzina programa nikada nije važnija od ispravnog programa,
- nakon što se postigao zadati cilj po pitanju funkcionalnosti koda, dizajna programa, postignute modularnosti kako bi se mogle raditi lake izmjene, onda se pristupa optimiziranju koda, a ujedno i provjeravaju performanse programa.

Entropija u izvornom kodu

Standardizovani pristup pisanju koda po pitanju jednostavnosti razumijevanja istog i lociranje „uskih grla“ koja ograničavaju prohodnost i brzinu izvođenja programskog koda je

posebna tehnika koju budući programeri moraju da usvoje. „Uska grla“ u aplikacijama su povezana s ključnim računarskim resursima koji koriste aplikacije. Kod dvoslojnih poslovnih aplikacija koje koriste bazu podataka mogu se uočiti dvije lokacije na kojima se uska grla najčešće javljaju:

- aplikacijski poslužitelj, koji je vezan za neefikasno pozivanje i konekciju sa bazama podataka,
- poslužitelj baze podataka, a koji je vezan za lošu logiku i normalizaciju baze.
- greške u kodu, a koje su vezane za kvalitet programskog koda, loša konfiguracija datoteka i loše strukture aplikacije.

Mjerenje

Mali dijelovi koda tokom izvođenja programa obično koriste neproporcionalnu količinu resursa i zbog toga uvijek treba izmjeriti programski kod da se nađe „usko grlo“. Nakon toga pristupa se optimiziranju koda na mjestima „uskog grla“. Poslije toga se ponovo mjeri brzina izvršavanja pod istim uslovima kao i prvi put prije optimiziranja. U rezultate optimiziranja koda ne možete biti sigurni dok se ne izmjeri vrijeme izvršenja koda. Mjerenja performansi moraju biti precizna [8].

Danas postoje mnogi kompajleri na nižem nivou koji mogu mjeriti brzinu izvršenja koda, a za složenije aplikacije postoje integrisani alati u okviru razvojnog okruženja kao što je profiler u okviru MS Visual Studija ili Java programskog okruženja [6]. Naravno jako je bitno da se prilikom mjerenja dijelova programskog koda integrira naredba ponavljanja koja će se izvesti preko sto hiljada puta, a opet zavisi od koda koji se mjeri, da bi dobili mjerljive rezultate.

Proces optimiziranja koda

Optimiziranje koda je repetitivan proces koji se upotrebljava kako bi se identificirala i eliminirala „uska grla“ u aplikaciji [8]. Poslije svakog skupa promjena testiramo i ponovno mjerimo kako bi vidjeli da funkcioniše i da vidimo rezultat svaki put koji se dobio optimiziranjem koda.

Razlozi smanjene efikasnosti koda

Prilikom pisanja koda često zbog šabloniziranja mogućeg rješenja dolazi do neefikasnosti koda, a što je slučaj kod početnika programera. Vrlo često isti stavljaju u drugi plan logiku i kreativnost u odnosu na „linija manjeg otpora“ kojoj se pribjegava da se ne troše u pronalaženju svog rješenja. Kako bi se izbjeglo smanjenje efikasnosti koda treba voditi računa o:

- input/output operacijama, prvenstveno treba da se izbjegne rad sa bazama podataka koje se nalaze na diskovima i uvijek kad je moguće da se radi sa bazama podataka/fajlovima u memoriji,
- straničenju, izbjegavati kada je moguće da se koriste operacije koje zahtijevaju od OS da mijenja stranice radne memorije zbog usporavanja izvršavanja,
- sistemskim pozivima, voditi računa o pozivima sistemskih rutina koje su veoma teške u većini slučajeva, jer uključuju snimanje stanja programa, povezivanje sa stanjem kernela itd.
- sopstvenim servisima, nastojati umjesto poziva sistemskih rutina da kreirate sopstvene servise koji su potrebi da zadovolje potrebe buduće aplikacije,

- sistemima, izbjegavati ulazak u sisteme u cilju povećanja neovisnosti aplikacije i poboljšanja performansi aplikacije, a u slučaju da je potrebno da aplikacija ulazi u sistem treba da se poveže sa proizvođačem OS u cilju stvaranja efikasnog poziva,
- interpretiranim jezicima, treba da se biraju oni koji ostvaruju manje gubitke u performansama zbog procesiranja programskih instrukcija prije kreiranja i izvršavanja mašinskog koda.

Sve su ovo razlozi smanjene efikasnosti koda i prilikom razvoja aplikacije treba voditi računa da se poštuju određena pravila, a da to ne umanji produktivnost programera i fokus programera treba da bude na funkcionalnom kodu.

Optimiziranje koda kao metoda učenja sintakse

Prilikom podučavanja budućih programera treba da se uvijek počne od toga da studenti razumiju gotov kod i da isti znaju analizirati po pitanju input/output. Ovaj način se pokazao korisnim u savladavanju sintakse jezika i razvijanju stilova pisanja koda. U anketi koja se sprovela za potrebe ovog rada među budućim softverskim inženjerima učestvovao je 81 student. Studenti su odgovarali online putem na sprovedenu anketu. Na pitanje: **"**Koji način vam najviše odgovara za učenje programiranja?**"**, odgovorili su kao što je prikazano na grafikonu:

Grafikon 1. Koji način vam najviše odgovara za učenje programiranja?

Iz grafikona 1. se može vidjeti da studenti prihvataju način učenja na tuđim kodovima i to je preduslov za prepoznavanje moguće optimiziranje koda.

Kako studenti postaju više iskusni u određenim znanjima oni proizvode sofisticiranije i složenije sisteme. Stručnjaci su skloniji stvaranju generalizacija, dok početnici mogu samo napraviti površno promatranje [2]. Koncepti programiranja koje studenti usvajaju razlikuju se svojom težinom od jednostavnijih kao što su varijable, strukture grananja i strukture petlji, zatim rada s ulaznim i izlaznim podacima, manipulacijom pogreškama pa sve do zahtjevnijih koncepata polja, strukturiranih tipova podataka, rekurzija, pokazivača i referenci [5].

Prvenstveno dobar pristup pisanju koda će proizvesti između ostalog i programere za testiranje koda, otklanjanje grešaka u kodu i na kraju programere koji će moći da optimiziraju kod. Na sljedećem primjeru koda koji je napisan u C# programskom jeziku ćemo pokazati kako se može optimizirati kod.

Prvo imamo funkciju koja nije optimizirana i koja će imati visoko vrijeme izvršenja zbog iteracija i korištenja nizova koji se sporije izvršavaju po pitanju procesiranja:

```
public void NotOptimisedFunction()
 {
    string prva_rijec = "Namir";
    string druga_rijec = "Aplikacija";
    Int64 integer_vrijednost = 160;

   if (prva_rijec.ToLower() != druga_rijec.ToLower())
     {
      for (int i = 0; i<integer_vrijednost; i++)
          {
          ArrayList intList = new ArrayList();
          intList.Add(i);
           if ( i % 10 == 0 & i % 20 ==0 )
              {
               Console.WriteLine("Uslov ispunjen");
              };
          };
        };
     }
```

Druga funkcija je optimizirana sa malim promjenama kodnih linija u prvoj funkciji kao što je if naredba, uvođenje while naredbe umjesto for i uvođenje listi umjesto nizova:

```
public void OptimisedFunction()
{
     string prva_rijec = "Namir";
     string druga_rijec = "Aplikacija";
     int integer_vrijednost = 160;
if (string.Compare(prva_rijec, druga_rijec, true) == -1)
  {
 while(integer_vrijednost>0)
       {
        List<int> intList = new List<int>();
        intList.Add(integer_vrijednost);

if(integer_vrijednost % 10 == 0 && integer_vrijednost % 20 == 0)
             {
            Console.WriteLine("Uslov ispunjen");
              };
             integer_vrijednost--;
       };
  };
}
```

Ove funkcije su pozvane u glavnoj funkciji kako bi se pokazalo vrijeme izvršenja aplikacije:

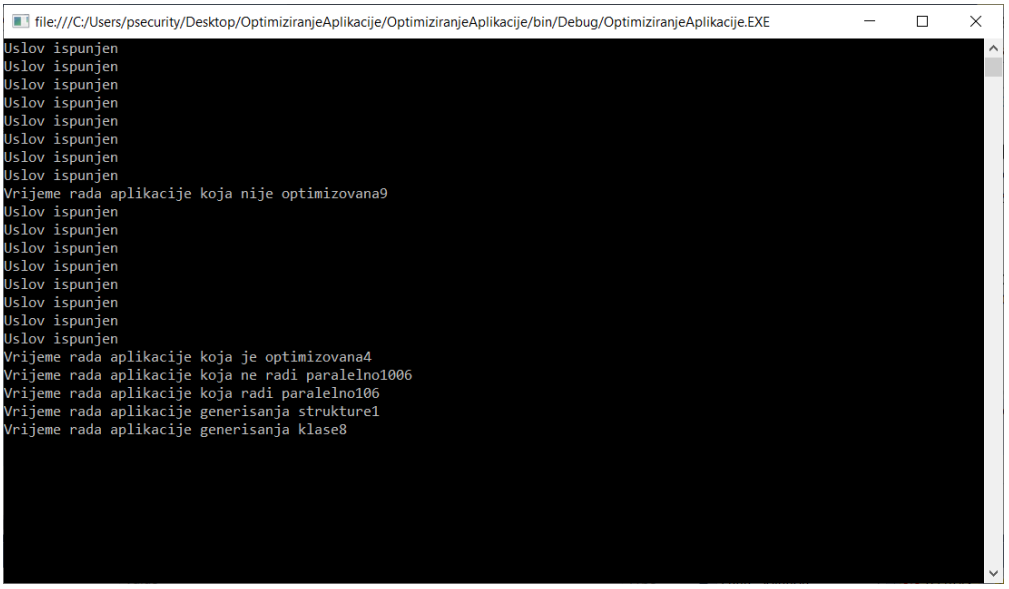
- koja poziva funkciju ne optimiziranu,
- koja poziva funkciju optimiziranu,
- kada radi paralelno,
- kada ne radi paralelno,
- kada generiše strukture,
- kada generiše klase.

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

```
using System.Threading.Tasks;

namespace OptimiziranjeAplikacije
{
   struct MojaStruktura
   {
      public string Ime;
      public string Prezime;
   }
   class MojaKlasa
   {
      public string Ime;
      public string Prezime;
   }
   class Optimiziranje
   {
      static void Main(string[] args)
      {
         var watch =
System.Diagnostics.Stopwatch.StartNew();
         new Optimiziranje().NotOptimisedFunction();
         watch.Stop();
         Console.WriteLine("Vrijeme rada aplikacije koja
nije optimizovana" + watch.ElapsedMilliseconds);
         watch = System.Diagnostics.Stopwatch.StartNew();
         new Optimiziranje().OptimisedFunction();
         watch.Stop();
         Console.WriteLine("Vrijeme rada aplikacije koja je
optimizovana" + watch.ElapsedMilliseconds);
         Console.ReadKey();
         watch = System.Diagnostics.Stopwatch.StartNew();
         for (int i = 0; i < 10; i++)
         {
         new Optimiziranje().DugoSeIzvrsava();
         }
         watch.Stop();
         Console.WriteLine("Vrijeme rada aplikacije koja ne
radi paralelno" + watch.ElapsedMilliseconds);
         watch = System.Diagnostics.Stopwatch.StartNew();
         for (int i = 0; i < 10; i++)
         {
    new Thread(new
Optimiziranje().DugoSeIzvrsava).Start();
         }
         watch.Stop();
         Console.WriteLine("Vrijeme rada aplikacije koja
radi paralelno" + watch.ElapsedMilliseconds);

         Console.ReadKey();
   MojaStruktura[] objStruct = new MojaStruktura[100000];
   MojaKlasa[] objClass = new MojaKlasa[100000];
      watch = System.Diagnostics.Stopwatch.StartNew();
      for (int i = 0; i < 100000; i++)
         {
            objStruct[i] = new MojaStruktura();
            objStruct[i].Ime = "Namir";
            objStruct[i].Prezime = "Neimar";
         }
      watch.Stop();
      Console.WriteLine("Vrijeme rada aplikacije
generisanja strukture" + watch.ElapsedMilliseconds);
      watch = System.Diagnostics.Stopwatch.StartNew();

      for (int i = 0; i < 100000; i++)
         {
            objClass[i] = new MojaKlasa();
            objClass[i].Ime = "Namir";
            objClass[i].Prezime = "Neimar";
         }
      watch.Stop();
      Console.WriteLine("Vrijeme rada aplikacije
generisanja klase" + watch.ElapsedMilliseconds);

      Console.ReadLine();
      }

      public void NotOptimisedFunction() {
         string prva_rijec = "Namir";
         string druga_rijec = "Aplikacija";
         Int64 integer_vrijednost = 160;

         if (prva_rijec.ToLower() != druga_rijec.ToLower())
         {
            for (int i = 0; i<integer_vrijednost; i++)
            {
               ArrayList intList = new ArrayList();
               intList.Add(i);
               if ( i % 10 == 0 & i % 20 ==0 )
               {
                  Console.WriteLine("Uslov ispunjen");
               };
            };
         };
      }

      public void OptimisedFunction()
      {
         string prva_rijec = "Namir";
         string druga_rijec = "Aplikacija";
         int integer_vrijednost = 160;
      if (string.Compare(prva_rijec, druga_rijec, true) == -1)
         {
            while(integer_vrijednost>0)
            {
               List<int> intList = new List<int>();
               intList.Add(integer_vrijednost);

       if(integer_vrijednost % 10 == 0 && integer_vrijednost
% 20 == 0)
               {
                  Console.WriteLine("Uslov ispunjen");
               };
               integer_vrijednost--;
            };
         };
      }
```

```
                                                    Thread.Sleep(1);
public void DugoSeIzvrsava()                            }
{                                                     }
   for (int i = 0; i < 1000; i++)                   }
   {                                              }
      if (i % 20 == 0)                         }
      {
```

Vrijeme izvršenja aplikacije za navedene slučajeve je prikazano na slici 1. i na osnovu rezultata koji će svaki put biti drugačiji prilikom novog pokretanja, može se zaključiti da je postignut cilj optimiziranja. Različiti rezultati u pogledu vremena izvršenja su prvenstveno vezani u pozivanju sistema jer se u ovoj aplikaciji koristio .NETFramework, Version=v4.5.2. Rezultati su prikazani u milisekundama. Vidljivo je na slici 1. da je vrijeme izvršenja aplikacije manje prilikom korištenja optimizirane funkcije, prilikom paralelnog rada aplikacije i pozivanja strukture u odnosu na ne optimiziranu funkciju, kada ne radi paralelno i kada poziva klasu.



Slika 1. Output izvršavanja aplikacije

Na osnovu analize programskog koda i izvršenog poređenja koda optimizirane i ne optimizirane funkcije studenti mogu da nauče kako i na koji način mogu poboljšati vrijeme izvršenja aplikacije, kao i sintaksu odgovarajućeg programskog jezika. Pokazalo se kroz anketu sprovedenu među studentima za potrebe ovog rada da im ovaj način odgovara u pogledu izgrađivanja programerskih vještina.

Zaključak

U analizi rada sa budućim programerima može se zaključiti da postoje ozbiljni problemi u primarnom i sekundarnom obrazovanju po pitanju izgrađivanja logičkog pristupa rješavanju problema i kreativnom pristupu rješavanja problema. Zbog toga je teže pronaći način na koji se može kod studenata razviti sopstveni pristup razvoju algoritama i stvaranja stila u programerskom svijetu bez obzira koji se programski jezik koristi. Jedan od načina prevazilaženja ovog problema kod studenata je analiza tuđeg koda i pronalaženja alternative u kodu po pitanju moguće optimiziranja koda. Ovo je izazov i pred nastavnicima da mijenjaju

tradicionalne navike podučavanja. Pored toga potrebno je da se koriste alati koji se zasnivaju na igrama i povezani sa vizualizacijom.

Naravno da postoje mnoge prediktorske varijable koje imaju uticaja na školovanje budućih programera kao što su motivacija, inteligencija, logičko razmišljanje, nastavni kadar, realni problemi koji se rješavaju, stilovi učenja, matematsko znanje, programske platforme za učenje prvog programskog jezika i povezanost prilikom učenja sa realnim sektorom u softverskoj industriji. Sve su to faktori koje treba ispitati u budućem istraživanju među studentima i onih koji su završili školovanje za programere. Praćenje karijera svršenih studenata je jako bitno kako bi sa vremenske distance mogli da identifikuju sve prediktorske varijable koje su imale uticaja na sticanje programerskih vještina.

Literatura

1. Ben-Ari, M. (1998), Constructivism in Computer Science Education, SIGSCE 98, Atlanta, GA, USA
2. Fincher, S. (1999), What are We Doing When We Teach Programming?, 29th ASEE/IEEE Frontiers in Education Conference, San Juan, Puerto Rico
3. Greenberg, I., Kumar, D., Xu, D. (2012), Creative Coding and Visual Portfolios for CS1, u SIGCSE 2012, raleigh, NS, USA
4. Kalid A. (2011), Understanding the Pareto Principle (The 80/20 Rule), http://betterexplained.com/articles/understanding-the-pareto-principle-the-8020-rule/
5. Piteira, M., Costa, C. (2012), Computer Programming and Novice Programmers, u ISDOC'12, Lisbon, Portugal
6. Robert P. (2011). Strategije i tehnike optimizacije programskog koda, diplomski rad, Sveučilište u Dubrovniku, https://bib.irb.hr/datoteka/565390.RP_dipl.pdf
7. Sam A. (2012), Dot Net Perls, http://www.dotnetperls.com
8. Steve M. (2004), Code Complete, Microsoft Press

CODE OPTIMIZATION IN THE FUNCTION OF DEVELOPMENT OF STUDENT PROGRAMMING SKILLS

Abstract

A proper approach to writing a program code and optimization is an important feature for a good programmer with regards to the software industry. The proper code writing, requires a foundation built on formal and in-formal education. The paper will show the proper code writing with the correct code optimisation in order to execute the application as quickly as possible. The contribution of the work is reflected in building skills in young people who are studying for future jobs as programmers. The goal is to enable future developers to create an image of how they can develop their own style of writing code.

Keywords: *functionality, errors, bottleneck, code optimization, paging, Pareto principle, interpreters, compilers, C, C#*