Simulacija sistema i dizajn programa

Slajdovi za predmet Osnove programiranja

Katedra za informatiku, Fakultet tehničkih nauka, Novi Sad

2022.

Ciljevi

- potencijalne primene simulacije kao sredstva za rešavanje realnih problema
- pseudoslučajni brojevi i njihova primena u Monte Carlo simulacijama
- primena top-down i spiralnih tehnika u pisanju složenih programa
- jedinično testiranje i otkrivanje grešaka u složenim programima

Simuliranje racquetballa

- simulacijom modelujemo realne procese
- na taj način možemo dobiti informacije koje bismo teško dobili na drugačiji način
- primene simulacije: vremenska prognoza, dizajn letelica, specijalni filmski efekti, itd.

Problem koji simuliramo

- Žika često igra racquetball sa protivnicima koji su malo bolji od njega
- Žika obično gubi mečeve!
- zar ne bi igrači koji su **malo** bolji trebali da pobeđuju **malo** češće?
- simulacijom možemo ustanoviti da li male razlike u sposobnostima uzrokuju velike razlike u rezultatima

Analiza i specifikacija

- racquetball igraju 2 igrača na terenu unutar 4 zida
- jedan igrač počinje igru **servisom**
- igrači se smenjuju udarajući lopticu reketom – **reli**
- reli se završava kada prvi igrač ne uspe da vrati lopticu u igru

Analiza i specifikacija $_2$

- ako je pogrešio igrač koji je servirao, servis preuzima drugi igrač
- ako je pogrešio igrač koji nije servirao, igrač koji je servirao dobija poen
- igrači mogu osvajati poene samo na svoj servis!
- prvi igrač koji osvoji 15 poena je pobednik

Analiza i specifikacija $_3$

- u našoj simulaciji sposobnost igrača se izražava kao verovatnoća da će igrač osvojiti reli na svoj servis
- primer: igrač sa verovatnoćom 0.6 će osvojiti 60% relija na svoj servis
- program će omogućiti unos verovatnoće za oba igrača i zatim simulirati više partija
- na kraju će ispisati rezultate simulacije

Ulazni podaci

- verovatnoće osvajanja poena za igrače A i B;
- broj partija koje će odigrati A i B

Izlazni podaci

- broj odigranih partija
- broj pobeda i procenat pobeda za igrače A i B

Napomene

- pretpostavljamo da su svi ulazni podaci ispravne numeričke vrednosti
- u svakoj simuliranoj partiji igrač A počinje prvi

Verovatnoća

- kada kažemo da igrač A pobeđuje u 50% relija, to ne znači da pobeđuje **fiksno** svaki drugi put
- više liči na bacanje novčića
- u proseku, u pola slučajeva će pasti glava, a u drugih pola će pasti pismo
- rezultat jednog bacanja novčića ne utiče na sledeće bacanje
- može se desiti da 5 puta uzastopno padne glava

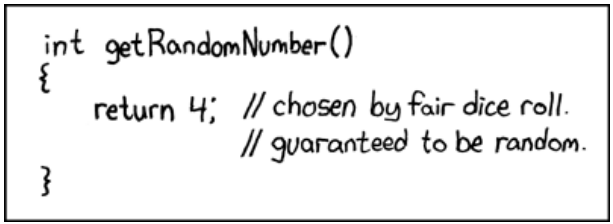
Verovatnoća

- veliki broj simulacija dovešće do toga da se događaji pojavljuju sa određenom verovatnoćom
- simulacije zasnovane na verovatnoći zovu se Monte Carlo simulacije

Slučajni i pseudoslučajni brojevi

- niz slučajnih brojeva: ne može se ustanoviti pravilo za određivanje narednog broja na osnovu prethodnog
- pseudoslučajni brojevi su nastali na osnovu preciznog pravila, ali čoveku **izgledaju** kao slučajni brojevi
- program `chaos.py` sa prvih predavanja: brojevi izgledaju nasumični, iako su izračunati po preciznoj formuli

Slučajni i pseudoslučajni brojevi



Generisanje pseudoslučajnih brojeva

- generator pseudoslučajnih brojeva funkcioniše na sličan način
- počinje od inicijalne (seed) vrednosti
- na osnovu nje se izračuna „slučajan“ broj
- kada se zatraži novi broj, prethodni broj se koristi u izračunavanju

Generisanje pseudoslučajnih brojeva $_2$

- niz brojeva izgleda nasumično
- ako pokrenemo generator sa istim **seed**-om dobićemo isti niz „slučajnih“ brojeva
- Python ima modul koji sadrži više funkcija za rad sa pseudoslučajnim brojevima
- ove funkcije izračunavaju **seed** na osnovu sistemskog datuma i vremena
- svaki put kada se pokrene, funkcija proizvodi različit niz brojeva

Funkcija randrange

- funkcija `randrange` vraća pseudoslučajan `int` iz datog opsega
- sintaksa je slična funkciji `range`
- `randrange(1, 6)`
  vraća „slučajno“ izabran broj iz `[1,2,3,4,5]`
- `randrange(5, 105, 5)`
  vraća „slučajno“ izabran broj iz `[5,10,15,20,...,100]`
- gornja granica za `randrange` ne ulazi u moguće vrednosti

Funkcija randrange 2

- svaki poziv randrange proizvodi jedan pseudoslučajan int

```
>>> from random import randrange
>>> randrange(1,6)
5
>>> randrange(1,6)
3
>>> randrange(1,6)
2
>>> randrange(1,6)
5
>>> randrange(1,6)
5
>>> randrange(1,6)
5
>>> randrange(1,6)
4
```

Funkcija randrange $_3$

- desilo se da se broj 5 pojavi više puta
- na velikom broju pokušaja dobićemo uniformnu raspodelu
- tj. sve vrednosti će se pojaviti otprilike jednak broj puta

Funkcija `random`

- funkcija `random` vraća pseudoslučajne `float` brojeve
- `random` nema parametre
- uvek vraća broj iz opsega $[0, 1)$

Funkcija random 2

```
>>> from random import random
>>> random()
0.79432800912898816
>>> random()
0.00049858619405451776
>>> random()
0.1341231400816878
>>> random()
0.98724554535361653
>>> random()
0.21429424175032197
>>> random()
0.23903583712127141
>>> random()
0.72918328843408919
```

random i racquetball

- simulacija racquetball partije koristi funkciju `random`
- ako je verovatnoća osvajanja poena za igrača 70% odnosno 0.7

```
if <igrač dobije na svoj servis>:
    score = score + 1
```

- treba nam izraz koji vraća `True` u 70% slučajeva

`random` i racquetball $_2$

- recimo da generišemo broj iz $[0, 1)$
- 70% tog intervala se nalazi levo od tačke 0.7
- u 70% slučajeva broj će biti manji od 0.7
- u preostalih 30% slučajeva će biti ≥ 0.7
- $=$ ide na gornji kraj intervala jer generator može da proizvede 0 ali ne i 1

random i racquetball 3

- ako prob predstavlja verovatnoću za igrača
- uslov random() < prob će biti zadovoljen sa traženom verovatnoćom

```
if random() < prob:
    score = score + 1
```

Top-down dizajn

- top-down dizajn predstavlja način za rešavanje problema
- složen problem se predstavi kao kombinacija jednostavnijih, manjih problema
- ovaj princip ponavljamo dok ne dođemo do problema koji su trivijalni za rešavanje
- rešenja malih problema zajedno čine rešenje velikog problema

Šablon ulaz-obrada-izlaz

- programi često koriste šablon ulaz-obrada-izlaz
- algoritam za simulaciju racquetball-a:

```
ispiši uvodni tekst
unesi podatke: probA, probB, n
simuliraj n partija korišćenjem probA i probB
ispiši rezultate simulacije
```

Top-down dizajn

- da li je ovo rešenje suviše apstraktno?
- sve što ne umemo da napravimo ignorisaćemo na trenutak
- pretpostavimo da su sve komponente koje nam trebaju da implementiramo algoritam već napravljene
- naš zadatak je tada jednostavan

Prvi korak

- prvo ispisujemo uvodni tekst
- ovo je lako, i nećemo da gubimo vreme na to

```
printIntro()
```

- pretpostavljamo da postoji funkcija printIntro() koja radi baš ono što nam treba!

Drugi korak

- u drugom koraku unosimo podatke
- i ovo je lako
- recimo da postoji funkcija `getInputs` napravljena za ovaj zadatak

```
printIntro()
probA, probB, n = getInputs()
```

Treći korak

- treći korak je simulacija n partija
- recimo da postoji funkcija `simGames` za ovaj zadatak
- koji ulazni podaci su njoj potrebni?
  - `probA`, `probB`, `n`
- koje rezultate ćemo dobiti od funkcije?
  - broj partija koje je dobio A
  - broj partija koje je dobio B

```
printIntro()
probA, probB, n = getInputs()
winsA, winsB = simGames(n, probA, probB)
```

Četrvrti korak

- četvtrti korak je ispis rezultata simulacije
- recimo da postoji funkcija printSummary koja obavlja taj posao
- koji ulazni podaci su njoj potrebni?
  - `winsA`, `winsB`

```
printIntro()
probA, probB, n = getInputs()
winsA, winsB = simGames(n, probA, probB)
printSummary(winsA, winsB)
```

Kompletan program

```
def main():
    printIntro()
    probA, probB, n = getInputs()
    winsA, winsB = simGames(n, probA, probB)
    printSummary(winsA, winsB)

main()
```

Razdvajanje zaduženja

- početni problem je dekomponovan na 4 podproblema
  - `printIntro`
  - `getInputs`
  - `simGames`
  - `printSummary`
- naziv, parametri i rezultat svake funkcije su već određeni
- to čini signaturu ili interfejs funkcije
  - preko parametara i rezultata funkcija „komunicira sa spoljašnjim svetom“

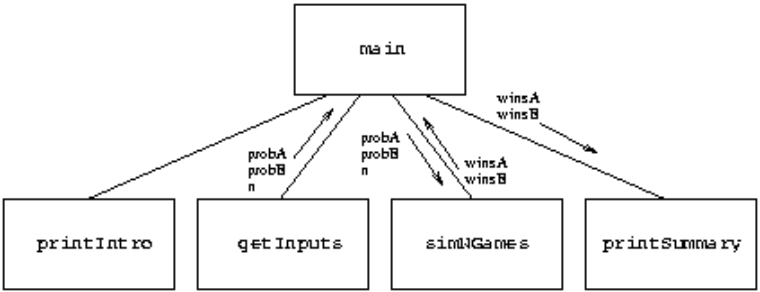
Razdvajanje zaduženja $_2$

- signature svih funkcija nam omogućavaju da nezavisno (paralelno) rešavamo te probleme
- kako radi `simGames` nije važno u funkciji `main`
- važno je samo da prima dogovorene parametre i vraća dogovoreni rezultat

Razdvajanje zaduženja $_3$

- strukturu našeg programa – tj. hijerarhiju možemo prikazati grafički
- svaka funkcija je pravougaonik
- linija između pravougaonika označava da funkcija iznad poziva (koristi) funkciju ispod
- strelice označavaju prenos podataka

Razdvajanje zaduženja 4



Razdvajanje zaduženja 5

- na svakom nivou dizajna interfejs nam govori koji detalji sa nižeg nivoa su bitni
- proces određivanja bitnih osobina i ignorisanje drugih detalja zove se apstrakcija
- top-down dizajn je postupak otkrivanja korisnih apstrakcija

Dizajn na drugom nivou: printIntro

- ponavljamo top-down pristup za svaki od podproblema koje smo već definisali
- funkcija `printIntro` treba da ispiše uvodnu poruku

```
def printIntro():
    print("Ovaj program simulira racquetball partije između igrača")
    print('zvanih "A" i "B".  Sposobnost svakog igrača je određena')
    print("verovatnoćom (broj između 0 i 1) da igrač osvoji poen")
    print("na svoj servis. Igrač A uvek servira prvi.\n")
```

- u drugoj liniji string je dat u jednostrukim navodnicima zato što želimo da ispišemo dvostruke navodnike
- nema novih funkcija → nema novih dijagrama

Dizajn na drugom nivou: getInputs

- funkcija getInputs unosimo tri broja koja vraćamo kao rezultat funkcije

```
def getInputs():
    a = eval(input("Verovatnoća da A osvoji poen na servis >> "))
    b = eval(input("Verovatnoća da B osvoji poen na servis >> "))
    n = eval(input("Broj partija za simulaciju >> "))
    return a, b, n
```

Dizajn na drugom nivou: `simGames`

- funkcija `simGames` simulira n partija i prati ukupan broj pobeda igrača
- „simulira n partija“ zvuči kao petlja
- „prati ukupan broj“ liči na akumulator

```
inicijalizuj winsA i winsB na 0
ponovi n puta
    simuliraj partiju
    if pobedio igrač A:
        inkrementiraj winsA
    else:
        inkrementiraj winsB
```

Dizajn na drugom nivou: simGames 2

- već imamo signaturu funkcije simGames

```
def simNGames(n, probA, probB):
    # simulira n racquetball partija između igrača A i B
    # vraća broj pobeda A, broj pobeda B
```

- početak je lak:

```
def simNGames(n, probA, probB):
    # simulira n racquetball partija između igrača A i B
    # vraća broj pobeda A, broj pobeda B
    winsA = 0
    winsB = 0
    for i in range(n):
```

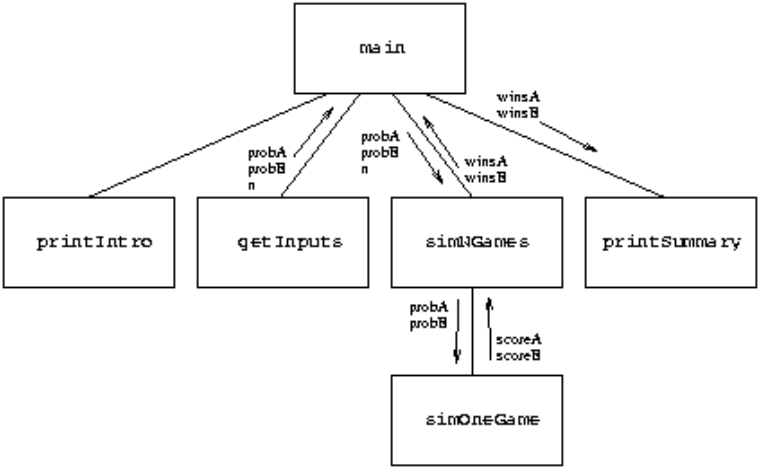
Dizajn na drugom nivou: `simGames` $_3$

- sledeći zadatak je simulacija jedne partije
- nismo sigurni kako to uraditi – ostavimo to za kasnije!
- recimo da postoji funkcija `simOneGame`
- ulazi za `simOneGame` su verovatnoće za igrače A i B
- šta je izlaz? – podatak ko je dobio partiju
- najlakše je preneti konačni rezultat (npr 15:11)
- pobednik je onaj sa više poena, njegov akumulator inkrementiramo

Dizajn na drugom nivou: simGames 4

```
def simNGames(n, probA, probB):
    # simulira n racquetball partija između igrača A i B
    # vraća broj pobeda A, broj pobeda B
    winsA = winsB = 0
    for i in range(n):
        scoreA, scoreB = simOneGame(probA, probB)
        if scoreA > scoreB:
            winsA = winsA + 1
        else:
            winsB = winsB + 1
    return winsA, winsB
```

Dizajn na drugom nivou: `simGames` 5



Dizajn na trećem nivou: `simOneGame`

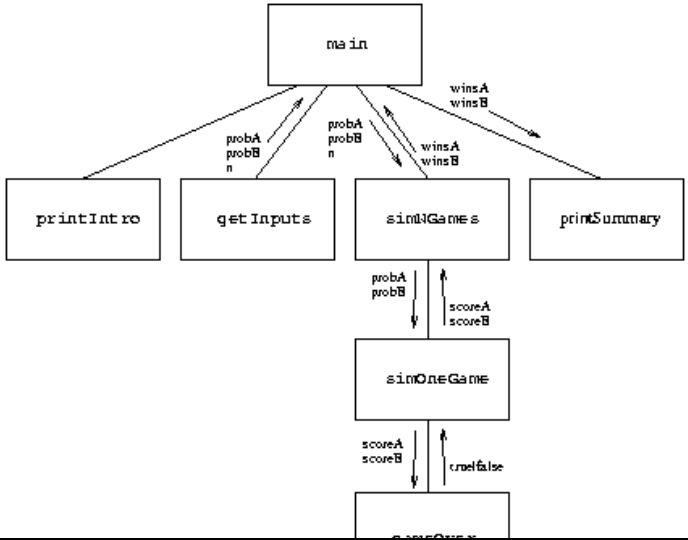
- igrači igraju relije dok jedan ne pobedi
- liči na beskonačnu petlju – ne znamo unapred koliko će relija trajati partija
- treba da pamtimo rezultat – dva akumulatora
- treba da pamtimo i ko servira – string koji uzima vrednosti `"A"` i `"B"`

Dizajn na trećem nivou: simOneGame 2

```
def simOneGame(probA, probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while <uslov>:
```

- šta će biti uslov?
- neka prosledimo rezultat(e) funkciji koja vraća True ako je partija gotova

Dizajn na trećem nivou: `simOneGame` 3



Dizajn na trećem nivou: `simOneGame` 4

- unutar `while` petlje treba da obradimo jedan servis
- poredićemo slučajan broj sa verovatnoćom igrača koji servira (`probA` ili `probB`)
- igrača koji servira pamtimo u promenljivoj `serving`

Dizajn na trećem nivou: `simOneGame` 5

- ako servira igrač A onda koristimo `probA`,
- zavisno od rezultata ili povećamo poene za A ili servis preuzima B

```
if serving == "A":
    if random() < probA:
        scoreA = scoreA + 1
    else:
        serving = "B"
```

Dizajn na trećem nivou: simOneGame 6

- ako servira B, radimo isto kao i u prethodnom slučaju, samo sa zamenjenim ulogama

```
if serving == "A":
    if random() < probA:
        scoreA = scoreA + 1
    else:
        serving = "B"
else:
    if random() < probB:
        scoreB = scoreB + 1
    else:
        serving = "A"
```

Cela funkcija simOneGame

```
def simOneGame(probA, probB):
    serving = "A"
    scoreA = 0
    scoreB = 0
    while not gameOver(scoreA, scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
             else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB
```

Dizajn na četvrtom nivou: gameOver

- za sada imamo signaturu funkcije gameOver

```
def gameOver(a,b):
    # a i b su poeni igrača
    # vraća True ako je partija gotova, inače vraća False
```

- igra je gotova kada bilo koji igrač osvoji 15 poena:

```
a == 15 or b == 15
```

Cela funkcija gameOver

```
def gameOver(a,b):
    # a i b su poeni igrača
    # vraća True ako je partija gotova, inače vraća False
    return a == 15 or b == 15
```

Cela funkcija printSummary

```
def printSummary(winsA, winsB):
    # Ispisuje rezultate simulacije
    n = winsA + winsB
    print("\nSimulirano partija:", n)
    print("Pobede A: {0} ({1:0.1%})".format(winsA, winsA/n))
    print("Pobede B: {0} ({1:0.1%})".format(winsB, winsB/n))
```

Rezime top-down pristupa

- počeli smo sa najvišeg nivoa strukture našeg programa
- na svakom nivou počeli smo uopštenim algoritmom i prevodili ga u precizan kod
- „rafinacija korak-po-korak“ (step-wise refinement)

Rezime top-down pristupa $_2$

1. izrazi algoritam kao seriju manjih problema
2. definiši interfejs (signaturu) za svaki od manjih problema
3. opiši algoritam u smislu interfejsa prema manjim problemima
4. ponovi postupak za svaki manji problem

Bottom-up implementacija

- iako smo bili pažljivi sa dizajnom, nema garancija da nismo napravili neke greške
- implementaciju je najbolje praviti u malim delovima (koracima)

Testiranje programa

- sistematično testiranje već od iole većeg programa:
- počnemo sa testiranjem na najnižem nivou strukture
- testiramo komponentu čim je završimo
- ne odlažemo testiranje za kasnije!

- možemo `import`-ovati naš program i pozivati njegove funkcije da proverimo da li rade ispravno

Primer testiranja

- možemo početi od funkcije gameOver

```
>>> import rball
>>> rball.gameOver(0,0)
False
>>> rball.gameOver(5,10)
False
>>> rball.gameOver(15,3)
True
>>> rball.gameOver(3,15)
True
```

Testiranje funkcije `gameOver`

- pokrili smo sve bitne slučajeve
- sa 0,0 simuliramo prvi poziv ove funkcije
- drugi test simulira stanje u sredini partije
- poslednja dva testa proveravaju pobedu oba igrača

Testiranje funkcije simOneGame

- sada možemo preći na testiranje funkcije simOneGame

```
>>> simOneGame(.5, .5)
(11, 15)
>>> simOneGame(.5, .5)
(13, 15)
>>> simOneGame(.3, .3)
(11, 15)
>>> simOneGame(.3, .3)
(15, 4)
>>> simOneGame(.4, .9)
(2, 15)
>>> simOneGame(.4, .9)
(1, 15)
>>> simOneGame(.9, .4)
(15, 0)
>>> simOneGame(.9, .4)
(15, 0)
>>> simOneGame(.4, .6)
(10, 15)
>>> simOneGame(.4, .6)
(9, 15)
```

Testiranje funkcije `simOneGame` 2

- kada su verovatnoće jednake, broj poena je „blizu“
- kada su verovatnoće značajno različite, rezultat je „čišćenje sa terena“

Jedinično testiranje

- testiranje svake komponente programa – jedinice – na ovaj način zovemo jedinično testiranje
- testiranje svake funkcije nezavisno olakšava pronalaženje grešaka
- pojednostavljuje testiranje celog programa

Rezultati simulacije

- da li je racquetball takva igra da male razlike u sposobnostima dovode do velikih razlika u rezultatima?
- ako Žika osvaja 60% poena na svoj servis, a njegov protivnik 5% više, koliko često će Žika pobeđivati?
- da probamo, tako što se Žikin protivnik servirati prvi

Testiranje celog programa

```
Ovaj program simulira racquetball partije između igrača
zvanih "A" i "B".  Sposobnost svakog igrača je određena
verovatnoćom (broj između 0 i 1) da igrač osvoji poen
na svoj servis. Igrač A uvek servira prvi.

Verovatnoća da A osvoji poen na servis >> .65
Verovatnoća da B osvoji poen na servis >> .6
Broj partija za simulaciju >> 5000

Simulirano partija: 5000
Pobede A: 3329 (66.6%)
Pobede B: 1671 (33.4%)
```

- sa malom razlikom u kvalitetu, Žika osvaja tek svaku treću partiju!

Druge metode za dizajn programa

- top-down nije jedina metoda za pisanje složenijih programa!

Prototip

- drugi način je da prvo napravimo uprošćenu verziju programa,
- i da postepeno dodajemo detalje dok ne dostignemo punu specifikaciju
- ova početna verzija programa zove se prototip

Spiralni razvoj

- pravljenje prototipova često podrazumeva spiralni proces razvoja programa
- umesto da razmatramo ceo problem u startu
  - kroz specifikaciju, dizajn, implementaciju, i testiranje
- napravićemo prototip
- koji ćemo unapređivati u više mini-ciklusa
- dok ne dođemo do konačne verzije programa

Prototipovi i spiralni razvoj

- kako smo racquetball simulator mogli napraviti spiralnim razvojem?
- napravimo prototip koji podrazumeva
  - 50-50 verovatnoću osvajanja svakog poena
  - igra se fiksno 30 relija
- kasnije dodajemo
  - pravilno dodeljivanje poena
  - promenu servisa
  - različite verovatnoće
  - itd.

Razvoj u više faza

- faza 1: početni prototip – 30 relija, 50-50 verovatnoća, ispis posle svakog servisa
- faza 2: dodavanje različitih verovatnoća za igrače
- faza 3: partija traje dok prvi igrač ne osvoji 15 poena; sada imamo simulaciju jedne partije!
- faza 4: simuliranje više partija, ispisuje se broj osvojenih partija
- faza 5: unos podataka sa tastature i formatirani ispis

Top-down vs spiralni razvoj

- spiralni pristup je koristan kada se srećemo sa nepoznatim sistemom, zahtevima, ili tehnologijom
- ako top-down ne funkcioniše, probamo spiralno!

Top-down vs spiralni razvoj

- spiralni razvoj nije suprotstavljen top-down dizajnu – oni se nadopunjuju
- kada pravimo prototip koristićemo top-down dizajn
- dobar dizajn je i kreativan proces i nauka
- nema striktnih pravila

- vežba stvara majstora!