Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/05BE03Ljubisic

RAZVOJ INFORMACIONOG SISTEMA ZA AUTO-ŠKOLE

DRIVING SCHOOL INFORMATION SYSTEM DEVELOPMENT

Jelena Ljubišić, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U ovom radu predstavljena je perspektiva C# programskog jezika kroz tri paradigme – objektno-orijentisanu, funkcionalnu i konkurentnu. Svaka paradigma je analizirana prema svojim opštim odlikama, a zatim je analizirano na koji način su te karakteristike primenjene u C# programskom jeziku.*

Ključne reči: *C# programski jezik, Objektno-orijentisana paradigma, Funkcionalna paradigma, Konkurentna paradigma*

Abstract – *This thesis represents the perspective of C# programming language through investigation of three paradigms - object-oriented, functional and concurrent paradigm. Each of these paradigm is analyzed and their general charachteristics are given. Furthermore, it is analyzed how are these charachteristics defined in C# programming language*

Keywords: *C# programming language, Object-Oriented Paradigm, Functional Paradigm, Concurrent Paradigm*

1. UVOD

Programska paradigma označava programski šablon ili način programiranja. Izučavanjem programske paradigme se upoznaju globalna svojstva jezika koji pripadaju toj paradigmi. Poznavanjem određene paradigme, olakšano se savlada svaki programski jezik koji toj paradigmi pripada.

Svaki programski jezik može podržati više paradigmi, što je i slučaj sa C# jezikom. On podržava i još neke paradigme pored onih koje su u fokusu ovog rada, međutim, izdvojene su tri najznačajnije i najzastupljenije paradigme, a to su objektna, funkcionalna i konkurentna.

C# programski jezik je pre svega objektno-orijentisan. Osnovni koncepti objektno-orijentisane paradigme – klase, objekti, svojstva, imenovani prostori, su neizostavni za druge dve paradigme, i koriste se u gotovo svim primerima koda koji su navedeni u radu.

2. OBJEKTNO-ORIJENTISANA PARADIGMA

Objektno-orijentisana paradigma je koncept progamiranja koja uključuje kreiranje objekata koji modeluju poslovni sistem. Modeluju se zajedno operacije i podaci sa kojima operacija radi i čuvaju se kao jedinstvena komponenta.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Aleksandar Kupusinac.

Objektno-orijentisana paradigma se zasniva na konceptima apstrakcije, enkapsulacije, nasleđivanja (generalizacije) i polimorfizma.

Apstrakcija objedinjuje osnovne koncepte koje neki entitet obezbeđuje sa ciljem rešavanja nekog problema. Softverski model treba da bude pojednostavljen model realnog sistema, ali sa dovoljno podataka koji ga opisuju.

Enkapsulacija je skup mehanizama koji obezbeđuje jezik za zaštitu od neželjenih izmena iz spoljnih izvora. U većini objektno-orijentisanih jezika, definisanjem privatnog pristupa promenljivoj u okviru klase se obezbeđuje da druge klase ne mogu da pristupe toj vrednosti direktno, već isključivo putem metoda za pristup i izmenu, što predstavlja skrivanje podataka. Enkapsulaciju karakteriše i skrivanje implementacionih detalja, upotrebom interfejsa. Preko interfejsa entitet nudi izvestan broj usluga i skriva implementacione detalje, koji za korisnika nisu bitni.

Nasleđivanje je mehanizam kreiranja novih od postojećih klasa. Izvedene klase imaju generalizovane karakteristike bazne klase, a pored njih i sopstvene specijalizovane karakteristike. To je važan princip objektno-orijentisane paradigme i naziva se generalizacijom i specijalizacijom.

Polimorfizam je kontekstno zavisno ponašanje, i usko je povezano sa konceptom nasleđivanja. Pored toga što izvedena klasa može da doda svoje specifično ponašanje, ona može i da izmeni ponašanje koje je nasledila od roditeljske klase kako bi njene specifičnosti bile uzete u obzir. U okviru polimorfizma su definisani pojmovi preopterećivanja i predefinisanja metoda ili operatora. Pored toga, postoji i parametarski polimorfizam, koji koristi upotrebu šablona metoda i klasa, čime se omogućava višestruka upotreba.

2.1 Osnovni koncepti objektno-orijentisane paradigme

Klasa je softverski dizajn koji opisuje generalna svojstva predmeta modeliranja. Za definisanje klase se koristi ključna reč *class*, nakon čega se navodi naziv klase. Klasa opisuje tip objekta, dok objekat predstavlja upotrebljivu instancu (primerak) klasa.

Polja i svojstva predstavljaju informacije koje objekti sadrže. Polja su poput promenljivih, jer se njihova vrednost može direktno čitati ili upisivati. Tradicionalne metode pristupa su u C# programskom jeziku zamenjene svojstvima, koji imaju efekat skrivenog modifikatora pristupa metoda. Standardna notacija za C# programski jezik je da se atribut obeležava nekim nazivom i piše se malim početnim slovom, a svojstvo se definiše nazivom istim kao i atribut i velikim početnim slovom.

To dozvoljava da se postavi vrednost atributa koristeći povezano svojstvo. Svojstvo omogućava da se koristi

sintaksa karakterična za polja, a ne metode kada se pristupa privatnom polju.
Konstruktori su metode klase koje se pozivaju se automatski kada se kreira instanca neke klase. Koriste se da bi se inicijalizovale vrednosti atributa date klase. Konstruktori moraju imati isti naziv kao i klasa koju instanciraju. Podrazumeva se da konstruktor nema povratnu vrednost, i ključna reč *void* se ne navodi. Tipovi konstruktora u C# programskom jeziku su: ugrađeni konstruktor, konstruktor sa parametrima, konstruktor kopije, privatni i statički konstruktor.
Modifikator pristupa opisuje domen pristupa objekta i njegovih članova. Da bi se specificirala dostupnost objekta ili člana, u njihovoj deklaraciji se specificira neki od dostupnih modifikatora pristupa. U C# programskom jeziku su dostupni sledeći modifikatori: *private*, *public*, *protected*, *internal*, *protected internal* i *private protected*. Modifikatori pristupa održavaju enkapsulaciju uvodeći restrikcije kroz kontrolu pristupa.

2.2 Nasleđivanje u C# programskom jeziku

Višestruko nasleđivanje nije podržano u C# programskom jeziku. Svaka klasa može imati najviše jednu eksplicitno navedenu baznu klasu. Grupa klasa koje su povezane nasleđivanjem formiraju strukturu hijerarhije klasa. Posmatrajući klase kroz hijerarhiju, na vrhu je uvek klasa koja je direktan potomak ugrađene klase *Object*. Na taj način, C# obezbeđuje da je sve klase nasleđuju.

U C#-u, potklasa specificira nasleđivanje od natklase simbolom ":", koji se navodi nakon navođenja naziva klase u njenoj definiciji. Nakon ovog simbola se navodi naziv klase koja se nasleđuje. Svaka klasa, bilo da je bazna ili izvedena, treba da enkapsulira svoju inicijalizaciju u konstruktoru. Svaka izvedena klasa treba da ima svoj konstruktor za specijalizovanu inicijalizaciju, ali često mora da pozove ponašanje baznog konstruktora.

Sve klase mogu biti nasleđene, osim ako se to eksplicitno ne specificira ključnom rečju *sealed*.

Apstraktne klase definišu svojstva i metode koje klase izvedene iz nje mogu da implementiraju. Ova klasa se može naslediti, ali se ne može instancirati. Klase koje se nalaze na vrhu hijerarhije su obično apstraktne klase. Može da sadrži potpunu implementaciju metoda koje definiše, može da definiše polja i konstante i deklaraciju konstruktora. Za specifikaciju se koristi ključna reč *abstract*. Značaj apstraktnih klasa je u održavanju generalizacije.

Interfejs je korisnički pogled na to šta neki entitet radi i razlikuje se od implementacije. Interfejs sadrži deklaraciju metoda, definiše dostupnost specificiranih operacija, bez implementacije istih. Za deklaraciju interfejsa koristi se ključna reč *interface*, nakon čega sledi ime interfejsa. Ne može sadržati promenljive, konstruktore ili kod sa implementacijom metode.

Klasa koja implementira dati interfejs treba da obezbedi implementaciju datih metoda u interfejsu. Svaka klasa može da implementira neograničen broj interfejsa. On enkapsulira znanje o objektu i predstavlja pogled sa korisničke strane. Implementacioni detalji entiteta se skrivaju interfejsom. Korisnik može da pristupi samo interfejsu i nema pristup podacima.

2.3 Polimorfizam u C# programskom jeziku

Ukoliko u baznoj i izvedenoj klasi postoje metode istog potpisa (isti naziv metode i isti ulazni parametri), onda je izvedena klasa zapravo predefinisala metodu iz bazne klase. Mogu se predefinisati samo metode koje su definisane u baznoj klasi kao virtuelne. U definiciji metode u baznoj klasi, pre definisanja povratnog tipa metode, dodaje se ključna reč *virtual*, čime se označava da je metoda virtuelna. Predefinisana metoda u izvedenoj klasi se obeležava u svojoj deklaraciji ključnom rečju *override*, pre nego što se navede povratni tip.
Preopterećivanje je pojam pridruživanja različitih operacija istom nazivu metode ili istom operatoru, a razlikuju se po prosleđenim parametrima. Odluka o tome koja se metoda poziva se donosi u fazi prevođenja, i to poređenjem tipova i broja argumenata.
Ideja predefinisanja i preopterećivanja metoda je da naziv metode nije više identifikator metode. Time se povećava fleksibilnost i robusnost softverskog sistema. Polimorfizam čini sistem otvorenijim za promene, jer se izmena sistema ovim olakšava.
Pored preopterećivanja i predefinisanja metoda, postoji parametarski polimorfizam ili generičko programiranje.
Generička metoda koristi parametrizovane tipove. Konkretan tip parametra se definiše kada se metoda poziva, i tada kompajler proverava da li je kod validan. Lista generičkih tipova podataka se nalazi u angular zagradi nakon navođenja naziva metode. Ukoliko postoji potreba za više generičkih tipova u metodi, odvajaju se zarezima. Po konvenciji pisanja C# koda se koristi veliko slovo "*T*" za definisanje generičkog parametra. Identifikator "*T*" može da predstavlja bilo koji tip podataka. Jedini uslov je da metoda mora da radi sa bilo kojim tipom podataka ili objektom koji mu je prosleđen.

3. FUNKCIONALNA PARADIGMA

Funkcionalna paradigma se zasniva na pojmu matematičkih funkcija. C# eksplicitno podržava neke koncepte funkcionalne paradigme, jer postoje brojne pogodnosti funkcionalnog programiranja: brže izvršavanje koda, čitljiviji kod, i pogodan je za kompleksna rešenja. Funkcija je ravnopravna sa ostalim tipovima podataka.
Izraz u funkcionalnom programiranju je reprezentacija vrednosti, a ne sama vrednost. Izračunavanje izraza je neophodna operacija, i njegovo izračunavanje se svodi na jednostavne supstitucije i redukcije.
U funkcionalnoj paradigmi nije dozvoljena dodela koja može da izazove, promenu stanja definisane promenljive, odnosno bočni efekat. Svaki put se u naredbi dodele kreira nova promenljiva. S obzirom da u funkcionalnoj paradigmi ne može biti dodeljena vrednost promenljivoj nakon njene inicijalizacije, umesto petlji se za iteracije koriste rekurzivne funkcije. Za dodelu vrednosti funkcijskoj promenljivoj koristi se dodela imenovane metode, ili delegati.

3.1 Delegati

Delegat je tip podatka koji u C# programskom jeziku predstavljaju referencu na metodu. Koriste se za prosleđivanje metode kao argumenta drugim metodama.
Ključna reč za inicijalizaciju delegata je *delegate*. Dostupni modifikatori pristupa za delegat su: *public*,

private, *internal* ili *protected*. Da bi se dodelila metoda delegatu, kreira se promenljiva delegatskog tipa koja ima potpis kompatibilan sa metodom koja mu se dodeljuje. Delegatski tip može koristiti generičke tipove kao parametre, čime se odlaže specifikacija jednog ili više tipova parametra ili povratnog tipa sve dok delegat nije inicijalizovan kao promenljiva.
U standardnim bibliotekama C# programskog jezika postoje ugrađeni delegatski tipovi. Takvi ugrađeni generički delegati su *Action* i *Func* delegati. *Action* ugrađeni delegat koji može da pokazuje samo na metode koje ne vraćaju parametre. *Func* delegat ima povratnu vrednost koja je generički predstavljena. Korišćenjem *Action* i *Func* delegata, kod je kraći, definicija delegata je lakša i brža.
Prava upotreba delegata je u izražavanju anonimnih funkcija koristeći lambda izraze.
U C# verziji 2.0 se javljaju anonimne metode, koje, obezbeđuju prečicu za kreiranje jednostavne i kratke semantike koja se upotrebljava samo jednom. Anonimne metode mogu biti korišćene i kao parametri druge metode.
C# se u verziji 3.0 proširuje sa lambda izrazima, kako bi kompletirali anonimne funkcije i obezbedili kratku notaciju koja je važna za njih. Lambda račun funkcije tretira kao izraze koji se postepeno transformišu do rešenja. U okviru lambda računa ne postoji koncept deklaracije promenljive i dodele naziva funkciji. Za transformacije se koriste redukcije i konverzije. Redukcije daju uputstva kako transformisati izraze iz početnog stanja u neko finalno stanje Lambda račun naglašava pravila za transformaciju izraza i ne zamara se arhitekturom mašine koja to može da ostvari.

3.2 LINQ

Sa C# 3.0 se uvodi još jedna osobina za *.NET framework*, a to je LINQ (*Language Integrated Query*), koja omogućava kreiranje i izvršavanje upita na kolekcijama podataka koje implementiraju *IEnumerable<T>* interfejs.
Generička klasu *IEnumerable<T>* ona predstavlja tip podataka za enumeraciju kolekcije podataka. To je interfejs tipa kolekcije, koji se primenjuje da bi se skupili svi podaci kolekcije koji su neophodni i da bi se enumerisali. Klasa *IEnumerable<T>* poseduje samo jedno svojstvo – *Current*, koje čuva elemente kolekcije za trenutnu poziciju enumeratora. Metode koje *IEnumerable<T>* klasa definiše su sledeće: *Reset*, *MoveNext* i *Dispose*.
LINQ upiti prepoznaju sekvence, koje predstavljaju objekte koji implementiraju *IEnumerable<T>*, i elemente, koji predstavljaju bilo koju stavku u sekvenci. Postoji više od pedeset operatora upita u *Enumerable* klasi uključenoj u *System.Linq* imenovanom prostoru.
LINQ implementira koncept odloženog izvršavanja (lenje evaluacije) kad se vrši upit nad podacima iz kolekcije, i neće izvršiti upit u konstruktorskom vremenu, nego u procesu enumeracije. Za enumeraciju se može koristiti *foreach* petlja, da bi se pozvala *MoveNext* komanda klase *IEnumerable<T>*, sa ciljem da se upit enumeriše. Lenja enumeracija je korisna kad je potrebno iterirati kroz beskonačnu petlju, i neće doći do prepunjavanja jer se *MoveNext* poziva samo kada je tražena. Čini da se kod izvršava brže jer kompajler ne mora da proveri sve logičke izraze ukoliko jedan od njih da rezultat.

4. KONKURENTNA PARADIGMA

Konkurentnu paradigmu karakteriše više procesa koji se izvršavaju u istom vremenskom periodu, a imaju zajednički cilj. Za konkurentno programiranje potrebna je podrška u okviru programskog jezika ili bibioteka. Osnovni koncepti su nit, proces i zadatak.
Programska nit predstavlja osnovnu jedinicu operativnog sistema koja konkuriše za procesorsko vreme. Svaka nit ima sopstvenu stek memoriju na kojoj čuva kopije svojih lokalnih promenljivih kojoj samo ta nit može da pristupi, i svoj sigurosni kontekst. Upotrebom niti se rešava problem propusnosti i odziva sistema.
C# programski jezik klasom *Thread* iz imenovanog prostora *System.Threading* obezbeđuje osnove za rad sa nitima, kao što je kreiranje i pokretanje niti. U C# programskom jeziku, svaki proces se pokreće izvršavanjem glavne programske niti koja se pokreće funkcijom *Main*. U toku izvršavanja procesa, moguće je kreirati proizvoljan broj niti iz bilo koje prethodno formirane niti.
Izvršna instanca svake aplikacije naziva se proces. Svaki proces poseduje resurse potrebne za njegovo izvršavanje. Procesi su međusobno u potpunosti izolovani, i za komunikaciju koriste međuprocesnu komunikaciju. Niti jednog procesa dele hip memoriju i memorijski prostor, i mogu direktno da komuniciraju.
Nit se smatra blokiranom kada se njeno izvršavanje pauzira iz bilo kog razloga. Nit blokirana odmah predaje procesorsko vreme koje je zauzimala, i od tog momenta prestaje sa korišćenjem deljenih resursa dok ne bude zadovoljen uslov zbog kojeg je blokirana. Metode za blokiranje su *Join* i *Sleep*. Stanje niti definisano u svojstvu *ThreadState* ove klase za blokiranu nit je *WaitSleepJoin.*

4.1 Sinhronizacioni problemi

Za pozive funkcija koje su međusobno zavisne i dele resurse procesa, neophodna je sinhronizacija. Program u kom se izvršava više programskih niti istovremeno nije bezbedan ukoliko se neki objekat ponaša neočekivano u ovakvom okruženju.
Problem uslova takmičenja je situacija kada dve ili više niti istovremeno pristupaju deljenim resursima, pri čemu ishod izvršavanja programa zavisi od toga koja je nit prva pristupila određenom programskom bloku. Time su vrednosti deljenih promenljivih su nepredvidive i slučajne. Primer ovog problema je ažuriranje vrednosti iste promenljive od strane više niti. Vrednost promenljive koja je upisana od strane poslednje niti koja joj pristupa će obrisati prethodna upisivanja od strane drugih niti. Za dve niti koje istovremeno upisuju vrednost neke promenljive, dešava se situacija da se te dve niti takmiče čija vrednost će biti poslednja upisana u datu promenljivu.
Koncept napredovanja programa (engl. *liveness*) je karakteristika programa koji se sekvencijalno izvršava. Njime se garantuje da program može da se završi i da se stalno pravi određeni progres u programu. U konkurentnoj sredini sa deljenim resursima, postoji veća mogućnost da dođe do toga da program ne može da nastavi sa radom, što za posledicu može da ima da program nikada ne završi svoj rad.

Da bi se sinhronizacioni problemi izbegli, potrebno je obezbediti da pristup deljenim resursima tako da bude zaštićen od nesinhronizovanog pristupa i izmena od strane različitih niti. Osim upotrebe metoda koje blokiraju niti, mogu se implementirati i sinhronizacioni mehanizmi. Sinhronizacija u ovom slučaju obezbeđuje da za sekvencu instrukcija, koja se naziva kritičnom sekcijom, obezbedi da se sve instrukcije kritične sekcije izvrše bez prekidanja.
Za kontrolisanje pristupu deljenim resursima, koristi se zaključavanje (katanci i muteks mehanizmi) ili uzajamno isključivanje (monitori i semafori).
Određeni mehanizam se koristi u zavisnosti da li niti koje pristupaju deljenom bloku dolaze iz istog ili različitog procesa. Interne niti su niti kreirane u okviru istog procesa, dok su eksterne niti kreirane iz različitih procesa.
Za sve navedene mehanizme, C# je obezbedio određene klase, koje definišu određene metode i svojstva kako bi obezbedile pouzdane sinhronizacione mehanizme.

4.2 Zadaci u asinhronom programiranju

Bazen objekata je šablon koji podrazumeva čuvanje inicijalizovanih objekata umesto da se objekti kreiraju i brišu na zahtev. Objekat se iz "bazena" uzima na zahtev, a nakon završetka rada nad tim objektom, umesto brisanja, objekat se vraća kako bi se mogao ponovo iskoristiti. *System.Threading.ThreadPool* je klasa C# programskog jezika koja implementira bazen za objekte tipa *Thread*. *ThreadPool* ima svojstvo *MaxThreads*, koje definiše maksimalan broj niti koje je moguće kreirati, i uobičajeno je da je broj niti manji nego što je broj zadataka koji čekaju na izvršavanje.
TaskQueue predstavlja red čekanja zadataka koji čekaju na izvršavanje redosledom kako su dodati u red i zauzimaju slobodne niti iz bazena niti. Nakon završetka izvršavanja zadatka, niti će biti dodeljen sledeći zadatak koji je na čekanju. Moguće je najviše jedan bazen na nivou procesa. U bazenu niti, metodom *QueueUserWorkItem()* se dodaje novi posao u red čekanja izvršavanja.
Šablon *Task-based asynchronous pattern* (dalje u tekstu skraćeno TAP) se koristi za proizvoljne asinhrone operacije u metodi i kombinuje status operacije i API koji je korišćen u interakciji sa ovim operatorima kako bi napravio jedinstven objekat. Ti objekti su *Task* i *Task<TResult>* tipovi, iz namespace-a *System.Threading.Tasks*. Objekat *Task* klase je centralna komponenta TAP šablona.
Task i *Task<TResult>* klase koriste niti koje se čuvaju u bazenu niti. *Task* klasa se koristi kada se metoda pokreće kao zadatak, a povratna vrednost nije potrebna. U suprotnom se koristi *Task<TResult>*.
Zadatak je jedinica programa koja može da se izvrši konkurentno sa drugim jedinicama istog programa. Zadaci ne moraju eksplicitno da budu pozvani. Kada program pokrene neki zadatak, ne mora uvek da čeka na njegovo izvršavanje pre nego što nastavi sa svojim. Kada se izvršavanje zadatka završi, kontrola ne mora da se vrati na mesto odakle je izvršavanje počelo.Svaki zadatak u programu može da bude podržan od strane jedne kontrolne niti ili procesa.
Pored upotrebe metode *StartNew*, koja kreira i pokreće zadatak, za kreiranje instance zadatka se najčešće koristi statička metoda *Run*. Pokreće zadatak koristeći podrazumevane vrednosti i bez zahtevanja dodatnih parametara.
Ključne reči *async* i *await* C# programski jezik objavljuje sa verzijom 5.0, kao važan dodatak za asihrono programiranje u ovom programskom jeziku. Uvode se u sam programski jezik, da bi asinhrono programiranje bilo jednostavno za implementaciju i čitanje. *Async* ključna reč se dodaje ispred imena metode, a *await* se koristi u implementaciji metode, obično metode *Run* klase *Task*. Kada tok izvršavanja dođe do *await* dela, nit koja je pozvala metodu će iskočiti iz metode i nastaviti dalje sa izvršavanjem negde drugde. Asinhroni kod onda zauzima drugu, odvojenu nit. Sav tok izvršavanja nakon *await* ključne reči je planiran da se izvrši kada je zadatak završen.

5. ZAKLJUČAK

U uvodu rada je napomenuto da je C# programski jezik definisan prvenstveno kao objektno-orijentisan jezik u opštim definicijama. Kroz rad je pokazano da on može da podrži i druge dve navede paradigme - funkcionalnu i konkurentnu. Veliki broj opisanih pojmova u svojoj osnovi ima objektne osobine, bez obzira kojoj paradigmi pripadaju.
Ovim se omogućava da C# programski jezik u svojoj osnovi zadrži objektnu-orijentisanost, koja predstavlja odlično rešenje za modelovanje sistema koji odgovara principima ove paradigme, kako bi se izgradio softverski sistem koji je održiv i spreman na izmene. Konkurentnom paradigmom se omogućava izgradnja aplikacija koje imaju brz odziv, koje mogu da obavljaju više procesa istovremeno bez lošeg iskustva za korisnike zbog prestanka rada aplikacije. Funkcionalnim programiranjem u aplikaciji se odziv sistema ubrzava. Sa jedne strane, razumljivo je C# okarakterisati kao objektno-orijentisan jezik, ali on obuhvata još mnogo karakteristika van ove paradigme, koje su definisane u radu, i koje C# čine programskim jezikom pogodnim za razvoj raznovrsnih aplikacija.

6. LITERATURA

[1] *S.Kendall, Object Orriented Programming using C#, first edition 2011, Ventus Publishing ApS,*
[2] *Materijali sa vežbi nastavnog predmeta Sigurnost i bezbednost u smart grid sistemima, master studija Primenjenog softverskog inženjerstva* (pristupljeno u maju 2019.)
[3] *Microsoft Developer Network,* https://msdn.microsoft.com/en-us/library
[4] *W.Anggoro, Functional C#, third edition, Packt Publishing, 2017*

Kratka biografija:

Jelena Ljubišić rođena je u Novom Gradu 19.04.1993. Školske 2011/2012. je upisala Fakultet tehničkih nauka u Novom Sadu. Zvanje diplomirani inženjer elektrotehnike i računarstva stekla je 2016. godine. Položila je sve predmete predviđene planom i programom na master akademskim studijama na odseku: Primenjeno softversko inženjerstvo.