Arhitektura Distribuiranih Sistema

Praktikum

Imre Lendak – lendak@uns.ac.rs

Mihalj Šagi – sagi@uns.ac.rs
Daniela Rosić – drosic@uns.ac.rs
Bojan Jelačić – bojan.jelacic@uns.ac.rs

Sadržaj

Cilj praktikuma

Praktikum ima za cilj da kroz zadatke pruži razumevanje distribuiranih sistema. Znanja neophodna za praćenje vežbi su:

- Objektno-orijentisano programiranje
- Osnovno znanje programskog jezika C#
- Računarske mreže

Praktikum dodatno pokriva proširivanje znanja programskog jezika C# programskim komponentama, koja su potrebna za razumevanje i implementaciju zadataka vezana za distribuirane sisteme.

Praktikum je organizovan u poglavlja, i svako poglavlje sadrži gradivo koje će biti izlagano na jednom terminu vežbi. U prilogu ovog praktikuma se nalazi i izvorni kod koji prati gradivo.

Nastavni materijali su dostupni na sledećim lokacijama:

- Iz računarskih laboratorija na FTN: \\nastava\nastavni materijal\ADS
- http://esi.ftn.uns.ac.rs/index.php/predmeti/89-distribuirani-sistemi

Vežba 1 – Osnove programskog jezika C#

Cilj ove vežbe je da pruži osnovno razumevanje *.NET framework* softverske platforme, razumevanje osnovnih koncepata *programskog jezika C#* koji je razvijen u .NET softverskom okruženju, kao i rad u *Microsoft Visual Studio 2013* razvojnom okruženju.

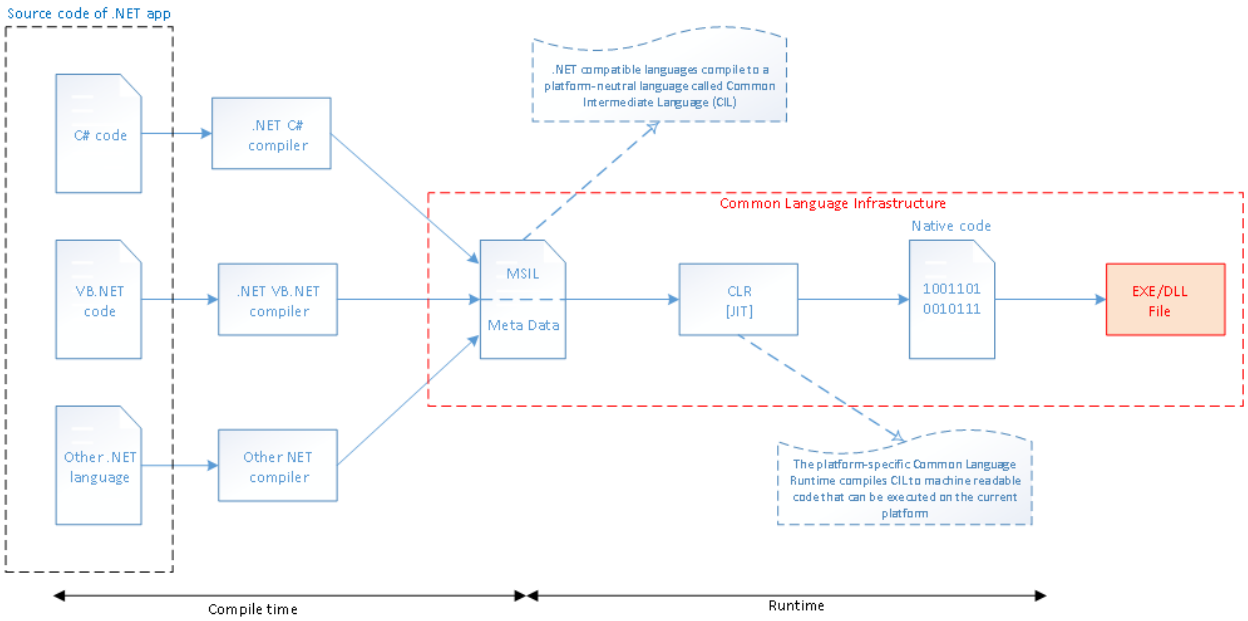
.NET Framework

.NET Framework predstavlja kontrolisano programsko okruženje za razvoj distrbuiranih aplikacija na Windows operativnom sistemu. .NET podržava više programskih jezika (Visual Basic, C++, C#, itd.) koji se izvršavaju u specifičnom softverskom okruženju, *Common Language Runtime*, koje u potpunosti kontroliše interakciju sa operativnim sistemom. Na taj način se obezbeđuje pisanje aplikacija nezavisno od platforme na kojoj će se aplikacija izvršavati. *Common Language Runtime (CLR)* je virtuelna mašina koja omogućuje kontrolisano izvršavanje programa pisanih za .NET, uključujući i kontrolu izvršavanja niti, upravljanja memorijom, rukovanje izuzecima i bezbednost aplikacije.

Kod koji se izvršava od strane CLR-a, odnosno u .NET runtime okruženju, naziva se *managed code*. Managed code je izolovan u smislu da mu se ne može pristupiti van runtime okruženja, niti se može pozvati direktno van runtime okruženja, što ga čini pouzdanijim i robusnijim u odnosu na*unmanaged code*. Za razliku od managed koda, *unmanaged code* podrazumeva direktnu interakciju sa operativnim sistemom što predstavlja veliki sigurnosni rizik.

.NET Framework uključuje i ogroman broj bibilioteka klasa (*Framework Class Library*) koje pružaju širok spektar mogućnosti, uključujući korisnički interfejs, pristup bazama podataka, razvoj distribuiranih i web aplikacija, podršku za konkurentno programiranje, itd. *.NET Framework Class Library (FCL)* su biblioteke nezavisne od programskog jezika, a organizovane su u logičke grupe, tzv. *namespaces*.

Na slici 1. je konceptualno prikazano *.NET runtime okruženje*. Prilikom kompajliranja, izvorni kod (pisan u nekom od .NET programskih jezika) se prevodi u *Common Intermediate Language (CIL)*, poznat i kao *Microsoft Intermediate Language (MSIL)*. CIL, odnosno MSIL, predstavlja skup instrukcija koji je nezavisan od platforme, a koji se može efikasno prevesti u mašinski/native code. Rezultat kompajliranja je *.NET assembly* koji enkapsulira sve podatke o aplikaciji, čime predstavlja osnovnu jedinicu za isporuku, verzionisanje i re-upotrebu aplikacije. *Just In Time (JIT) compiler* konvertuje MSIL code u skup mašinskih instrukcija u toku izvršavanja. CLR obezbeđuje različite JIT kompajlere, specifične za određenu arhitekturu. .NET CLR rezultuje DLL ili EXE fajlom.



Slika 1. .NET Runtime Okruženje

Osnovni koncepti C# programskog jezika

C# je objektno-orijentisani programski jezik razvijen u okviru .NET softverskog okruženja. U nastavku su objašnjeni osnovni koncepti kako bi se bolje razumele osnove programiranja u C#. Kao razvojno okruženje koristi se Microsoft Visual Studio 2013.

Kreiranje novog C# projekta u Visual Studio

- Pokrenuti Visual Studio (VS - verzija 2013)
- Kreirati novi C# projekat:
  - File → New → Project → Izabrati Visual C# project template
  - U ovom zadatku treba implementirati *konzolni aplikaciju*
- Podesiti projekat u zavisnosti od platforme na kojoj se pokreće VS:
  - Configuration Manager → Podesiti aktivnu konfiguraciju projekta, kao i platformu
    - Debug konfiguracija
    - Release konfiguracija
  - Project properites → Build → Output path

C# Programska struktura

Struktura svakog C# programa sastoji se iz sledećih komponenti:

1. **Namespace deklaracija.**
   Namespace je način organizacije različitih klasa u logičke grupe, čime se omogućuje kontrola opsega klasa i metoda u okviru većih programskih projekata. Da bi se u

programu koristile klase iz određenog namespace-a, navodi se ključna reč **using** kojom se definiše da dati program koristi imena iz navedenog namespace-a. Prva linija **using System;** označava da će*System* namespace biti uključen u program. Moguće je navesti više *using* izraza. Nakon toga sledi definicija namespace-a u okviru projekta. Definicija namespace-a počinje ključnom reči **namespace**, nakon čega se navodi ime namespace-a.

```
namespaceime_namespace
{
// deklaracija koda
{
```

2. **Deklaracija klase.**
   Podrazumeva deklaraciju promenljivih i metoda klase. Osnovna klasa, **class Program**, sadrži jednu statičku metodu – metoda **Main**. Metoda main predstavlja ulaznu tačku za svaki C# program.
   Preimenovati klasu Program u **MyFirstCSharpProgram**. Obratiti pažnju da prilikom preimenovanja VS nudi mogućnost preimenovanja promenljive/metode na nivou celog projekta kako bi se sprečile potencijalne greške.

3. **Komentari.**
   Komentari treba da budu sastavna komponenta svakog programa. Iako kompajler ignoriše linije koda koje predstavljaju komentar, treba ih navoditi radi lakšeg razumevanja napisanog koda. U zavisnosti od potrebe, komentari se mogu definisati na različite načine:

```
// one-line comment
```

```
/* first line comment
second line comment
…
the last line comment */
```

```
/// description of class / method summary
```

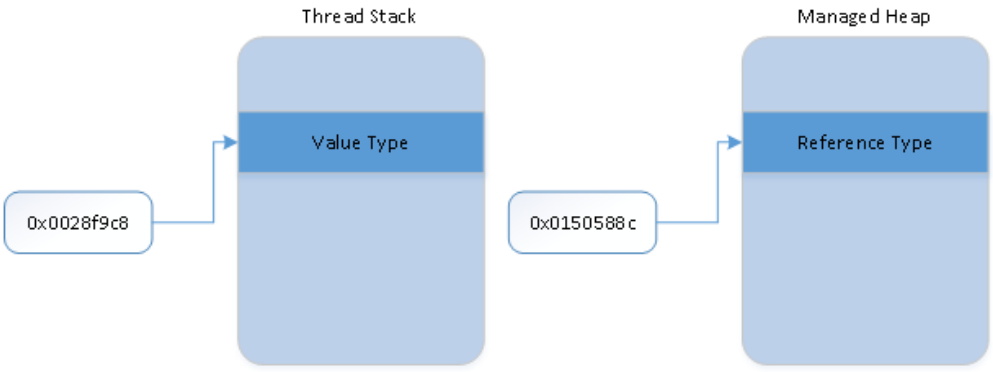
4. **Izrazi i iskazi kojima se definiše ponašanje programa.**
   Klasa **Console** definisana u okviru *System* namespace-a je klasa za ispis na konzolu, kao i čitanje sa konzole.

Zadatak 1. Ispisati "Hello World!" na konzoli. Obezbediti čekanje na pritisak tastera sa konzole kako bi program nastavio sa radom. Ispisati vrednost prethodno unetu kroz konzolu od strane korisnika.

Osnovni tipovi podataka i osnovni operatori u C#

Tip (*type*) je osnovna jedinica programabilnosti u .NET. Postoje dve kategorije tipova podataka:

1. ***Value types.*** Value type su tipovi podataka nastali iz klase **System.ValueType**. Tu spadaju: primitivni tipovi podataka (*int, double, bool, char, itd.*), strukture (*struct*) i enumeracije (*enum*). Promenljivim ovih tipova podataka vrednosti se dodeljuju direktno, odnosno svaka promenljiva sadrži direktno dodeljenu vrednost. Ukoliko se prilikom definisanja promenljive ne navede inicijalna vrednost, biće dodeljena default vrednost u zavisnosti od tipa. Ovi tipovi podataka se smatraju relativno malim podacima koji se čuvaju direktno na stacku aktivne niti.
2. ***Reference types.*** Reference type su tipovi podataka koji ne sadrži vrednost promenljive, već predstavljaju referencu na memorijsku lokaciju promenljive. Više promenljivih ovog tipa mogu da pokazuju na istu memorijsku lokaciju, što implicira da će se izmena vrednosti od strane jedne promenljive reflektovati i na druge promenljive. Promenljive ovog tipa se skladište u okviru posebne memorijske strukture (*heap*) i u potpunosti je kontrolisano od strane *Garbage Collectora (GC)*. Treba imati u vidu da je skladištenje podataka na heap-u i njihova kontrola od strane GC-a je skupa operacija. Iz tog razloga je objekte koji su relativno mali i kratkog životnog veka (short-lived) efikasnije skladištiti na stacku niti u okviru koje se kreiraju.



Slika 2. Skladištenje Value i Reference tipova podataka

Zadatak 2. Prodiskutovati *result = x / y* u zavisnosti od tipa promenljivih (*int vs double*).

```
int res; double a = 7, b = 3;
res = a/b; →Cannot convert type 'double' to 'int'.
```

Eksplicitno kastovanje daje na znanje programskom prevodiocu u kom formatu želimo da tumačimo vrednost promenljive. Ukoliko ne kastujemo eksplicitno, kompajler će implicitno da kastuje u trenutku kada to bude neophodno. Treba voditi računa da nije moguće kastovati vrednosti tipa koji ima veći opseg u vrednost tipa koji ima manji opseg (*npr. double vrednost u int*).

Zadatak 3. Prodiskutovati razliku između prefix i postfix inkrementa (**++x** vs **x++**). Šta je rezultat izvršavanja sledećeg programa?

```
inta, b, x = 10;
a = ++x;
b = x++;
Console.WriteLine("a={0}, b={1}, x={2}", a, b, x);
```

1. Prefix inkrement vraća vrednost x nakon ažuriranja (uvećanje za 1) lokacije promenljive x
2. Postfix inkrement vraća vrednost promenljive x, a zatim ažurira lokaciju te promenljive (uvećava za jedan).

Grananja u C#

Grananje, odnosno *decision making structure*, je iskaz koji podrazumeva izvršavanje različitih izraza u zavisnosti od toga da li su određeni uslovi ispunjeni ili ne, odnosno da li je vrednost uslova tačna (*true*) ili netačna (*false*). Tipične strukture grananja u C# su navedene ispod.

1. **IF struktura grananja.** Ukoliko je *Boolean* izraz tačan, kod unutar if bloka će biti izvršen. Ukoliko je *Boolean* izraz netačan, kod unutar if bloka će biti preskočen, i program će nastaviti sa prvom sledećom naredbom van if bloka.

```
if (boolean_izraz)
{// kod koji će biti izvršen ukoliko je boolean_izraz tačan }
```

2. **IF ... ELSE struktura grananja.** Ukoliko je Boolean izraz tačan, kod unutar if bloka će biti izvršen. Ukoliko je Boolean izraz netačan, kod unutar else bloka će biti izvršen.

```
if (boolean_izraz)
{   // kod koji će biti izvršen ukoliko je boolean_izraz tačan
}
else
{ // kod koji će biti izvršen ukoliko je boolean_izraz netačan
}
```

3. **IF ... ELSE IF struktura grananja.** Struktura koja se koristi ukoliko postoji više različitih uslova u zavisnosti od kojih će se izvršavati različiti programski blokovi. Kada se naiđe na prvi tačan izraz u okviru if uslova, ostali uslovi se neće proveravati. U slučaju da se poslednji else izraz ne navede, ukoliko nijedan od navedenih izraza nije bio tačan, nastavlja se sa prvom sledećom naredbom van ove strukture.

```
if (boolean_izraz_1)
{   // kod koji će biti izvršen ukoliko je boolean_izraz_1 tačan }
elseif (boolean_izraz_2)
{ // kod koji će biti izvršen ukoliko je boolean_izraz_2 tačan }
…
else { // kod koji će biti izvršen ukoliko nijedan od prethodnih izraza nije tačan }
```

4. **SWITCH-CASE struktura grananja.** Omogućuje izvršavanje određenog programskog bloka na osnovu provere jednakosti promenljive (odnosno switch izraza) sa listom vrednosti koje su definisane kao konstante tipa kome taj izraz odgovara (a koje se nazivaju *case*). *Default*blok je opcioni. Ukoliko postoji, a nijedan prethodni iskaz nije tačan, kod u okviru default bloka će biti izvršen.

```
switch (izraz)
{
case const_izraz_1: {kod koji se izvršava ako je izraz= const_izraz_1}; break;
case const_izraz_2: {kod koji se izvršava ako je iskaz= const_izraz_2}; break;
  …
default {kod koji se izvršava ukoliko nijedan od prethodnih izraza nije tačan}
}
```

Zadatak 4. Napisati program koji proverava da li je uneti odgovor (*string answe*r) potvrdan (“da” ili “yes”, dozvoljene su sve kombinacije malog i velikog slova), i u skladu sa tim postaviti promenljivu *bool isAffirmative* na odgovarajuću vrednost. Napomena: Za poređenje string vrednosti koristiti metode klase **System.String**.

Zadatak 5. Umesto if-else iskaza, zadatak 4. implementirati korišćenjem conditional operatora.

t ?x : y – ukoliko je iskaz t tačan, izračunaj i vrati vrednost x; u suprotnom, izračunaj i vrativrednost y

Forma conditional operatora je:

Zadatak 6. Za IF-ELSE iskaz u primeru ispod napisati ekvivalentan switch-case iskaz. Uporediti brzinu izvršavanja *if-else* u odnosu na ekvivalentan *switch-case* iskaze.

```
int a;
if (a == 1) {Console.WriteLine("Case 1"); }
elseif (a == 2) {Console.WriteLine("Case 2"); }
..
elseif(a == 5) {Console.WriteLine("Case 10"); }
else{Console.WriteLine("Unknown value"); }
```

Ukoliko swith-case iskaz sadrži više od pet case blokova, implementira se koristeći lookup tabelu ili hash listu. Na taj način se obezbeđuje da je za svaki case izraz potrebno isto vreme pristupa, za razliku od liste if-else izraza gde se izrazima pristupa redosledom kojim su i navedeni. To znači da pre nego što se dođe do poslednjeg if-else izraza potrebno je proći kroz sve prethodne izraze, čime se može znatno produžiti vreme izvršavanja koda.

Petlje u C#

Petlje (*loops*) predstavljaju kontrolne strukture koje obezbeđuju da se određeni programski blok izvrši određeni broj puta. Tipične kontrolne strukture, odnosno petlje u C# su navedene ispod.

1. **FOR petlja.** Sintaksa for petlje je data u nastavku. ***Init*** je korak koji se izvršava prvi put, i samo jednom. U ovom koraku mogu se deklarisati i inicijalizovati promenljive kontrolne strukture, npr. brojači. ***Condition*** je uslov koji se računa u svakom prolazu kroz petlju i ukoliko je rezultat tačan, telo petlje će se izvršiti. U suprotnom, telo petlje se neće izvršiti, a program nastavlja sa prvom naredbom nakon for petlje. ***Increment*** je skup izraza koji se izvršava nakon što je izvršeno telo petlje, npr. da bi se ažurirala vrednost brojača. Nakon ovog koraka, ponovo se proverava uslov zadat kroz *condition* i sve dok je ta vrednost tačna, telo petlje će se izvršavati.

```
for (init; condition; increment)
{ // kod koji se ponavlja}
```

2. **WHILE petlja.** Sintaksa while petlje je data u nastavku. While petlja je kontrolna struktura koja obezbeđuje ponavljanje programsog bloka sve dok je ***condition*** tačan. Na ovaj način, ukoliko je pre početka izvršavanja while petlje uslov netačan, telo petlje se neće izvršiti nijednon.

```
while(condition)
{ // kod koji se ponavlja}
```

3. **DO-WHILE petlja**. Za razliku od FOR ili WHILE petlji koje pre izvršavanja tela petlje proveravaju da li je uslov zadovoljen, DO-WHILE petlja proverava uslov nakon izvršavanja tela petlje. Na taj način se garantuje da će telo petlje biti izvršeno najmanje jednom, a izvršavaće se sve dok *condition* ne postane netačan. Sintaksa DO-WHILE petlje u C# je data u nastavku.

```
do
{ // kod koji se ponavlja}
while(condition);
```

Break i Continue naredbe

Break i **continue** naredbe su iskazi koji menjaju sekvencijalni redosled izvršavanja programa. Kada se ***break iskaz*** koristi u telu petlje, on zaustavlja izvršavanje tela petlje, i transferuje izvršavanje programa na prvu naredbu nakon petlje. ***Continue iskaz*** prekida izvršavanje tela petlje, i transferuje izvršavanje programa na proveru uslova iste petlje, a u zavisnosti od rezultata uslova petlje, telo petlje će se izvršiti ili će program izaći iz petlje.

Beskonačne petlje

Beskonačne petlje su petlje u kojima uslov nikada neće postati netačan, i samim tim izvršavanje programa nikada neće izaći van tela petlje. Tipičan način pisanja beskonačne petlje je: **for (; ;)**. Naime, kada uslov nije naveden podrazumeva se da je on tačan. Ukoliko je potrebno, inicijalizacija i inkrement u okviru petlje se mogu navesti, ali bez navedenog uslova petlja je beskonačna.

Zadatak 7. Napisati program koji na korisničkoj konzoli prikazuje zvezdice (*) kao na slici.

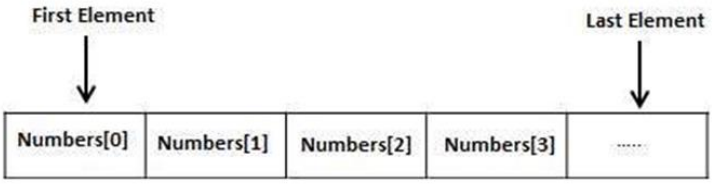
```
*
**
***
****
*****
******
*******
********
*********
**********
```

Nizovi u C#

Niz je sekvencijalna kolekcija elemenata istog tipa. Umesto deklarisanja svake promenljive posebno (*npr. int num0, int num1, ... int numN*), moguće je definisati niz promenljivih (istog tipa) kojima se pristupa preko indeksa. Na primeru ispod definisan je niz promenljivih tipa int, i način na koji se promenljivim tog niza pristupa:

```
int Numbers[] = new array[N+1];
num0 → Numbers [0] ... numN → Numbers [N]
```

Niz predstavlja kolekciju promenljiih istog tipa smeštenih u susedne memorijske lokacije – najniža memorijska adresa odgovara prvom elementu, a najviša adresa odgovara poslednjem elementu niza.



Slika 1. Zauzimanje memorije prilikom inicijalizacije niza

Deklarisanje i inicijalizacija niza

Deklarisanje niza: ***datatype[] arrayName;*** gde [] definiše rank niza, odnosno njegovu veličinu. Deklarisanje niza ne podrazumeva i njegovu inicijalizaciju, odnosno zauzimanje memorijskog prostora za elemente tog niza. Niz je reference type i prilikom inicijalizacije potrebno je koristiti ključnu reč **new**. Primer inicijalizacije niza: ***double[] Numbers = new double[10];***Za pristupanje ili dodelu vrednosti elementima niza se koristi index niza: ***Numbers[0] = 4500.5;***

Zadatak 8. Napisati program koji inicijalizuje niz od deset elemenata tipa int tako da vrednost svakog elementa niza odgovara vrednosti indeksa tog elementa u nizu uvećana za 100. Nakon toga, potrebno je ispisati sve elemente tog niza.

Zadatak 9. Izmeniti način iteriranja kroz niz koristeći **foreach** petlju.

S obzirom da niz predstavlja kolekciju podataka, za prolazak kroz sve elemente niza moguće je koristiti foreach petlju. Sintaksa foreach petlje u C# je data ispod. Dakle, izvršavanje tela petlje se završava kada se prođe kroz sve elemente niza (u opštem slučaju, unesto niza može biti bilo koja kolekcija podataka. Slično kao i kod ostalih petlji, i izvršavanje tela foreach petlje moguće je prekinuti naredbom break, nakon čega program nastavlja sa izvršavanjem prve sledeće naredbe van petlje.

```
foreach (tip_elemenata_niza var in ime_niza)
{ // kod koji se ponavlja }
```

Multidimenzionalni nizovi

U C# je moguće definisati i takozvane multi-dimenzionalne nizove. Najjednostavnija forma multidimenzionalnog niza je 2-dimenzionalni niz, a koji predstavlja niz čiji su elementi 1-dimenzionalninizovi. 2-dimenzionalni niz se može zamisliti kao matrica, odnosno tabela sa X vrsta (na slici 1) i Y kolona (na slici 2). Elementima ovakvog niza (matrice) se pristupa preko dva indeksa, i i j, na sledeći način: **a [i, j]**.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Slika 2. Primer dvodimenzionalnog niza sa 3 vrste i 4 kolone

Zadatak 10. Proizvoljno inicijalizovati 2-dimenzionalni niz sa N=10 vrsta i kolona, i sabrati sve elemente niza ispod glavne dijagonale (uključujući i samu dijagonalu).

Vežba 2 – Klase i osnovni koncepti objektno orijentisanog programiranja

Cilj vežbeje da pruži razumevanjepojma klase i osnovnih koncepata objektno orijentisanog programiranja.

Klase u C#

Klasa predstavlja strukturu podataka koja omogućuje definisanje novog tipa koji enkapsulira ponašanje objekata iste klase. Kroz klasu se definišu promenljive i metode koje opisuju ponašanje objekata date klase. Objekat predstavlja instancu klase koji zauzima određen memorijski prostor prilikom inicijalizacije. Klasa je reference type tako da se objekti skladište na heap-u.

Definicija klase upočinje ključnom reči **class**, zatim ime klase, nakon čega sledi definicija članova klase (odnosno promenljiivh i metoda koji čine klasu). Ime klase postaje novi tip koji se može koristiti za kreiranje objekata klase (odnosno promenljivih koje su tipa te klase). Kreiranje, odnosno inicijalizacija objekata se obavlja u konstruktoru klase. Ukoliko se konstruktor ne definiše eksplicitno, default konstruktor bez parametara se podrazumeva.

Specifikator pristupa (*access specifier*) definiše nivo pristupa kako za objekte određene klase, tako i za članove klase:

- ***Public:*** članovi klase definisani kao public su vidljivi i van te klase, odnosno dostupni su izvan klase kako u okviru istog assemblija tako i iz drugih assemblija koji referenciraju taj.
- ***Private:*** članovi klase definisani kao private su zaštićeni od spoljnog pristupa, i njima mogu pristupati isključivo funkcije članice iste klase.
- ***Protected:*** članovi klase kojima mogu pristupati funkcije članice iste klase, ali i iz klase nastale nasleđivanjem te klase.
- ***Internal:*** članovi klase koji su dostupni samo u okviru istog assemblija.
- ***Protected internal:*** članovi klase koji su dostupni u okviru istog assemblija, ali i iz klasa definisanih u drugom assembliju ukoliko nasleđuju datu klasu.

Default nivo pristupa je *private*. Članovima klase se pristupa navođenjem tačke (.) nakon imena objekta čijem članu se pristupa.

Zadatak 1. Definisatiklasu *Rectangle* koja opisuje pravougaonik. Objekat pravougaonika je opisan određenom dužinom (*doublewidth*) i širinom (*doublelength*). Default objekti ove klase treba da imaju dužinu i širinu 1, ali trebaomogućiti kreiranje objekta proizvoljnih dimenzija. Za objekte klase *Rectangle* treba obezbediti izračunavanje površine i obima objekta.

Property

U C#, polje klase (*field*) je promenljiva bilo kog tipa definisana u okviru klase. **Property** je član klase kojim se obezbeđuje pristup poljima te klase, odnosno kojim je moguće obezbediti

manipulisanje (čitanje, pisanje) poljima. Praksa za pisanje lepog i održivog koda je da su polja klase uvek definisana kao *private*, i da se izlažu spoljašnjem svetu preko Property-a. Property-u se, kao i ostalim članovima klase, može pristupati u zavisnosti od definisanog specifikatora pristupa. Sintaksa Property-a u C# je data ispod. *Property Ime* obezbeđuje manipulisanje *poljem ime* van klase.

```
classStudent
{
privatestring ime;
  ...
Publicstring Ime
  {
get { returnime; }
set {ime = value; }
}
}
```

Zadatak 2. Obezbediti javni pristup privatnim poljima klase *Rectangle*.

Statički članovi klase u C#

Kada je član klase definisan kao statički (ključna reč **static**), to znači da bez obzira koliko je objekata ove klase instancirano, postoji tačno jedna kopija tog člana klase zajednička za sve objekte. Statičkim članovima klase se može pristupati bez obzira da li postoji instanciran objekat te klase. Statičke promenljive se uglavnom koriste za definisanje konstanti koje su zajedničke za sve objekte date klase. Statičke metode mogu pristupati isključivo statičkim poljima klase, i takođe mogu se pozivati i pre kreiranja objekata. Ukoliko je klasa definisana kao statička, to znači da nije moguće instancirati objekte ove klase. Statičkim članovima klase se pristupa koristeći ime klase, umesto imena konkretnog objekta klase.

Zadatak 3. Implementirati statičku metodu u okviru klase *Rectangle* koja vraća broj ukupno kreiranih objekata ove klase. U okviru main metode pozvati ovu metodu i ispisati povratnu vrednost.

Interfejsi

Interfejs (*interface*) definiše šta treba da sadrži svaka klasa koja nasleđuje taj interfejs. Interfejs sadrži deklaraciju članova klase (property-i i metode), bez implementacije istih. Od svake klase koja nasleđuje dati interfejs se očekuje implementacija članova navedenih u interfejsu. Za deklaraciju interfejsa koristi se ključna reč **interface**, nakon čega sledi ime interfejsa. Kao nepisano pravilo, imena interfejsa uvek počinju velikim slovom 'I'.U C# je moguće naslediti samo jednu klasu, dok je moguće naslediti više interfejsa.

Zadatak 4. Napisati interfejs koji odgovara klasi *Rectangle*implementiranoj kroz prethodne primere. Dodatno, u interfejsu definisati i novu metodu *void ShowInfo()* koja treba da ispiše podatke o dimenzijama objekata *Rectangle*.

Osnovni koncepti OOP

Osnovni koncepti objektno orijentisanog programiranja (OOP) su:

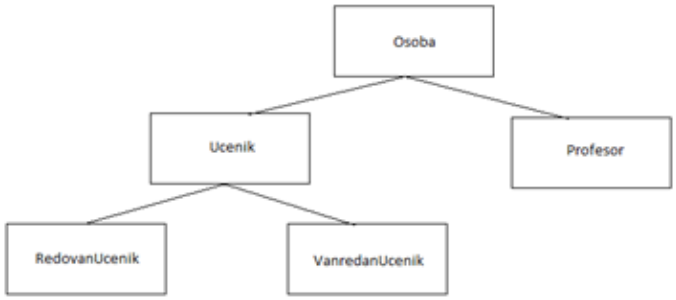
1) enkapsulacija (*encapsulation*),

2) nasleđivanje (*inheritance*),

3) polimorfizam (*polymorphism*).

Nasleđivanje je mogućnost da se na osnovu postojećih klasa izvedu nove klase koje treba da prošire, iskoriste ili izmene ponašanje definisano u postojećim klasama. Klasa koja se nasleđuje se naziva bazna klasa (*baseclass*), a koja nasleđuje drugu klasu se naziva izvedena (*derived class*). Nasleđivanje može biti jednostruko ili višestruko. C# podržava jednostruko nasleđivanje, odnosno jedna klasa može imati najviše jednu baznu klasu. Jednu baznu klasu može da nasledi više izvedenih klasa, čiji broj može biti neograničen.

Grupa klasa koje su povezane nasleđivanjem formiraju strukturu koja se naziva *hijerarhija klasa*. Dubina hijerarhije predstavlja broj nivoa nasleđivanja. Preporuka je hijerarhija bude najviše dubine 7. Na slici 1. je dat primer hijerarhije dubine 2. Hijerarhiju čini bazična klasa *Osoba*, koju nasleđuju klase *Učenik* i *Profesor*. Klasa *Učenik* ima dve implementacije, *RedovanUčenik* i *VanredanUčenik*.



Slika 3. Hijerarhija uloga

U primeru ispod prikazana je sintaksa nasleđivanja. Javni članovi bazne klase su implicitno i javni članovi izvedene klase. Nasleđivanje ne podrazumeva da će izvedena klasa imati pristup svim članovima bazne klase. Privatni članovi bazne klase, iako su nasleđeni, su dostupni isključivo članovima bazne klase.Članovi bazne klase sa modifikatorom protected su jedino dostupni unutar bazne klase i direktno i indirektno izvedenim klasama. Ukoliko modifikator pristupa nije naveden pretpostavlja se da je *private*.Dostupnost izvedene klase je uslovljena dostupnošću bazne klase. Ako je bazna klasa privatna, izvedena ne može biti javna. Osim korišćenja implementiranih funkcionalnosti ili implementacije apstraktnih metoda, u izvedenoj klasi se mogu dodati i novi članovi, specifični samo za datu klasu. Za poziv konstruktora bazne klase iz konstruktora izvedene klase se koristi ključna reč **base**.

```
publicclassBaznaKlasa
{
publicvoid PosaljiPoruku(string poruka)
    {
    }
}
publicclassIzvedenaKlasa : BaznaKlasa
{
//Konstruktor izvedene klase
public IzvedenaKlasa()
    {
    }

//Poziv konstruktora bazne klase iz izvedene klase
publicIzvedenaKlasa : base()
    {
    }
}
IzvedenaKlasa izvedenaKlasa = newIzvedenaKlasa();
izvedenaKlasa.PosaljiPoruku("poruka");
```

Polimorfizam se zasniva na ideji da metoda, deklarisana u osnovnoj klasi, može biti implementirana na više različitih načina u različitim izvedenim klasama.Polimorfizam se realizuje preko virtualnih metoda. Za deklaraciju virtualnih metoda se koristi ključna reč **virtual**. Virtualne metode se mogu re-implementirati u izvedenim klasama, kada se koristi ključna reč **override**. Pritom, virtualna i re-implementirana metoda moraju imati 1) isto ime 2) isti modifikator pristupa 3) isti tip rezultata 4) iste tipove parametara. Sintaksa virtualne i re-implementirane metode može se videti u primeru ispod.

```
publicclassBaznaKlasa
{
publicvirtualvoid PosaljiPoruku(string poruka)
    {

    }
}
publicclassIzvedenaKlasa : BaznaKlasa
{
publicoverridevoid PosaljiPoruku(string poruka)
    {
string novaPoruka = “Izvedena klasa” + base.PosaljiPoruku(poruka);
    }
}
```

Ukoliko ne želimo da drugi programeri nasleđujuklasukoju implementiramo, to možemo učiniti pomoću ključne reči **sealed**, a takve klase se nazivaju *zapečaćene klase*. Definicija takve klase data je u primeru ispod.

```
publicsealedclassZapecacenaKlasa
{
}
```

Apstraktne klase su koncept usko povezan sa interfejsima. Za ove klase se ne mogu instancirati objekti, mada za razliku od interfejsa (koji nikad ne sadrže implementaciju), apstraktne klase mogu delimično biti implementirane. Ključna razlika između interfejsa i apstraktne klase je činjenica da klase mogu implementirati neograničen broj interfejsa, dok najviše jednu apstraktnu klasu mogu naslediti. Klasa koja nasledi jednu apstraktnu, i dalje može implementirati neograničen broj interfejsa. Značaj apstraktnih klasa se ogleda u tome što je neke funkcionalnosti moguće implementirati kao zajedničke za sve klase naslednice, dok se druge metode specifične samo za klase naslednice mogu naknadno implementirati kada i sama klasa naslednica. Dakle, svaka konkretna klasa koja implementira interfejs mora implementirati sve njegove metode, dok se u slučaju apstraktne klase neke članice mogu implementirati,a neke članice se deklarisati kao apstraktne i implementirati u klasi naslednici.

Deklaracija apstraktne klase u C# počinje ključnom reči **abstract**, nakon čega sledi standardna C# deklaracija klase. Takođe, metode za koje se očekuje da budu implementirane u klasama naslednicama se označavaju ključnom reči **abstract**. Apstraktna klasa je klasa koja se može naslediti, ali se ne može instancirati. Može posedovati apstraktne metode koje nemaju svoju implementaciju, ali ih klase naslednice moraju implementirati korišćenjem ključne reči **override**.

Zadatak 5. Napisati program koji omogućava uplatu i isplatu na devizni i tekući račun, kao i ispis stanja računa nakon datih uplata i isplata. Provizija na uplatu na tekući račun je 0, a provizija na isplatu je 3%. Provizija na uplatu i isplatu sa deviznog računa je 5%. I devizni i tekući račun su definisati svojim jedinstvenom brojem (broj računa), kao i početnim stanjem računa.

Vežba 3 – Kolekcije podataka

Cilj ove vežbe je da pruži napredno znanje u radu sa različitim tipovima kolekcija podataka u okviru programskoj jezika C#.

Liste u C#

Lista predstavlja C# kolekciju podataka istog tipa kojima se pristupa preko indeksa. Odnosno, lista predstavlja dinamički niz ili vektor. Pristupanje elementima niza preko indeksa je brza operacija. Prilikom dodavanja elemenata u listu, korisnik ima potpunu kontrolu kada je u pitanju redosled elemenata te liste. Međutim, dodavanje i uklanjanje elemenata sa početka ili sredine liste je prilično skupa operacija, jer podrazumeva shiftovanje svih elemenata na stacku up/down u zavisnosti da li se element dodaje ili uklanja.

Inicijalizacija liste

Defaut inicijalizaciju liste podrazumeva kapacitet liste 0. U situaciji kada je potrebno dodati novi element u listu, a kapacitet liste je dostignut, veličina liste se udvostručava. To znači da ukoliko je default kapacitet liste 0, prilikom dodavanja prvog elementa lista će biti reinicijalizovana na 4, kada dostigne kapacitet biće ponovo reinicijalizovana na 8, itd. U slučaju liste sa velikim brojem elemenata ovo može uzrokovati nepotrebno zauzimanje memorijskog prostora.

```
// Default inicijalizacija liste:
List<int> listObj = newList<int>();
```

Dakle, prilikom inicijalizacije liste treba voditi računa da se lista inicijalizuje sa unapred definisanim kapacitetom kad god je to moguće. Na taj način se znatno poboljšava rukovanje memorijom. Inicijalizacija liste sa unapred definisanim kapacitetom sledi:

```
// Inicijalizacija liste sa unapred definisanim kapacitetom:
List<int> listObj = newList<int>(157345);
```

Kad god su elementi liste poznati unapred, najbolja praksa je definisati vrednosti tih elemenata prilikom inicijalizacije liste, umesto kasnije pozivanja metode *Add*. Dakle, umesto:

```
List<int> listObj = newList<int>();
listObj.Add(5);
listObj.Add(15);
listObj.Add(25);
```

elemente liste treba inicijalizovati na sledeći način:

```
List<int> listObj = newList<int>() {5, 15, 25};
```

Metoda Contains

Metoda Contains vraća vrednost true/false u zavisnosti od toga da li se element naveden kroz parametar nalazi u listi ili ne. Međutim, treba voditi računa da korišćenje ove metode može uticati na performanse aplikacije iz razloga što se vreme pretrage koristeći ovu metodu povećava linearno sa povećanjem kapaciteta liste. Preporuka je da se ova metoda koristi isključivo za male kolekcije podataka.

HashSet u C#

HashSet je C# struktura podataka koja, slično kao i liste, sadrži set objekata istog tipa, ali je za razliku od liste optimizovana za efikasniju pretragu elemenata. Naime, objekti se u HashSet strukturu skladište tako da indeks odgovara hashcode-u objekta koji se dodaje. Na taj način se onemogućuje skladištenje dva ista objekta u HashSet, kao i brza detekcija da li element postoji u HashSet-u. Iz tog razloga, da bi se optimizovale pretrage elemenata u listama velikog kapaciteta, preporuka je da se umesto liste koristi HashSet struktura podataka. (U nekim situacijama se ppokazalo da je pretraga *Contains vs HashSet* imala odnos 70 min naspram nekoliko milisekundi).

Negativna strana čuvanja podataka u HashSet strukturi je otežan prolazak kroz strukturu, kao i pristupanje elementima HashSet-a. Naime, HashSet je neuređena kolekcija podataka (*unordered*), odnosno prilikom skladištenja neće biti sačuvan redosled dodavanja elemenata, jer se elementi skladište na osnovu svog hashcode-a.

Zadatak 1. Inicijalizovati listu integer vrednosti od ukupno 154357 elemenata. Vrednost svakog elementa liste inicijalizovati na sledeći način:

- ukoliko je broj iteracije deljiv sa 4, element liste treba da odgovara broju iteracije,
- u suprotnom, element liste treba da odgovara broju iteracije sa negativnim predznakom.

Proći kroz sve elemente liste i ispisati zbir elemenata liste koji su deljivi sa 17.

Zadatak 2. Datu listu prepakovati u HashSet strukturu podataka.

Prilikom prepakivanja podataka iz jedne strukture u drugu treba voditi računa da se to radi u okviru konstruktora strukture u koju se vrši prepakivanje, umesto pisanjem petlji i prepakivanjem elemenata jedan po jedan.

Zadatak 3. Verifikovati performanse prethodno kreiranih *HashSet vs List* kolekcija prilikom pretrage da li se element -100001 nalazi u svakoj od kolekcija. Pretragu ponoviti 100 000 puta u slučaju obe kolekcije i uporediti dobijena vremena.

Dictionary u C#

Dictionary je možda i najčešće korišćena kolekcija u C# koja predstavlja asocijativni kontejner. Asocijativna iz razloga što se prilikom skladištenja podacima dodeljuje ključ (*key*) koji služi za manipulisanje podacima u kolekciji. ***Dictionary<Tkey, TValue>*** važi za najbržu asocijativnu kolekciju jer u osnovi koristi *HashTable* strukturu. To znači da su vrednosti ključeva hash vrednosti, čime se znatno poboljšavaju performanse u radu sa ovom vrstom kolekcija. Razlika između *HashTable* i *Dictionary* kolekcije je u tome što je dictionary generički tip, što ovu kolekciju čini *type safe* strukturom, odnosno nije moguće dodati slučajan tip elementa u kolekciju, a samim tip nije potrebno kastovanje podataka prilikom čitanja elemenata iz kolekcije. Vreme potrebno za dodavanje, uklanjanje i pretraga je relativno konstantno bez obzira na veličinu kolekcije.

Inicijalizacija Dictionary kolekcije

Slično kao i liste, dictionary je treba inicijalizovati sa unapred definisanim kapacitetom kad god je to moguće u cilju poboljšanja rukovanja memorijom.

```
                    //Umesto default inicijalizacije dictionary:
Dictionary<long, object> dictObj = newDictionary<long, object>();

// Inicijalizacija dictionary sa unapred definisanim kapacitetom:
Dictionary<long, object> dictObj = newDictionary<long, object>(157354);
```

Try-Get metoda

```
Dictionary<int, long> dictObj = newDictionary<int, long>(100);
dictObj.Add (5, 135.6);

// Pristupanje elementu dictionary-a preko ključa.
//U slučaju da ključ ne postoji biće bačen izuzetak KeyNotFoundException.
long temp = dictObj[5];
```

Elementima dictionary-a se pristupa preko vrednosti ključa:

Ukoliko nismo sigurni da li se ključ po kom pristupamo elementu dictionary-a zaista nalazi u dictionary-u, potrebno izvršiti odgovarajuću proveru pre pristupanja. Ta provera se najčešće izvršava koristeći metodu *ContainsKey*, kao što je navedeno:

```
if (dictObj.ContainsKey(5))
{
long a = dictObj[5];
. . .
}
```

Međutim, ukoliko postoji potreba za čestim pristupanjem elementima saverovatnoćom da ključ postoji u dictionary-u, efikasnije je koristiti metodu *TryGetValue*. Ova metoda u pozadini proverava da li navedeni ključ postoji u kolekciji, a zatim vraća vrednosti elementa koji je dodeljen tom ključu. U slučaju da ključ ne postoji, biće vraćena default vrednost koja odgovara tipu elementa. Dakle, za pristupanje elementima preko ključa treba koristiti metodu *TryGetValue* kao mnogo pouzdaniji i efikasniji način pristupanja:

```
long a;
if (dictObj.TryGetValue(1, out a))
{
 . . .
}
```

Zadatak 4. Verifikovati performanse *TryGetValue* vs *ContainsKey* metoda prilikom pristupanja elementu dictionary kolekcije preko ključa. Pristupanje ponoviti 1000000 puta, jednom u slučaju da ključ postoji u kolekciji, a zatim u slučaju da ključ ne postoji u kolekciji.

Poboljšanje performansi kada se koristi *TryGetValue* metoda je očiglednije što je broj uspešnih pronalazaka elemenata po ključu veći. Razlog je način implementacije *TryGetValue* metode koji podrazumeva poziv metode *ContainsKey*, a zatim i pretragu niza u slučaju da ključ postoji. Ispod je navedena implementacija obe metode dobijena dekompajliranjem:

```
publicbool TryGetValue(TKey key, out TValue value)
{
int index = this.FindEntry(key);
if (index >= 0)
 {
value = this.entries[index].value;
returntrue;
 }
value = default(TValue);
returnfalse;
}

publicbool ContainsKey(TKey key)
{
return (this.FindEntry(key) >= 0);
}
```

Add or Update metoda

Ukoliko je potrebno, u zavisnosti od toga da li ključ već postoji u kolekciji, odgovarajući par *<key, value>* dodati (*add*) ili ažurirati (*update*) najčešći pristup u implementaciji je provera da li određeni ključ postoji:

```
if (!dictObj.ContainsKey(5))
{
dictObj.Add(5, 135.6);
}
else
{
        dictObj[5] = 135.6;
}
```

Efikasniji način da se ovo implementira je samo pristup kolekciji preko indeksa, što u pozadini radi dodavanje ili ažuriranje u zavisnosti od toga da li ključ postoji u kolekciji:

```
dictObj[5] = 135.6;
```

Dictionary čija je vrednost tipa liste

Jedan od problema kada je u pitanju dictionary kolekcija čija je vrednost tipa liste je dodavanje elementa u listu, u zavisnoti od od toga da li određeni ključ postoji u dictionary-u. Najčešći pristup je provera da li ključ postoji, i u zavisnoti od toga se nova lista prvo inicijalizuje:

```
Dictionary<long, List<object>> dictObj = newDictionary<long, List<object>>(10);
. . .
if (!dictObj.ContainsKey(13))
{
        dictObj.Add(13, newList<object>(1));
}
List<object> objList = dictObj[13];//vraća listu iz kolekcije čiji je ključ 13, ili
dictObj[13].Add(…);  //dodavanje novog elementa u listu iz kolekcije čiji je ključ 13
```

Efikasniji način kako da se ovo implementira je:

```
Dictionary<long, List<object>> dictObj = newDictionary<long, List<object>>(10);
. . .
List<object> objList = null;
if (!dictObj.TryGetValue(1, out objList))
{
        objList = newList<object>(1);
        dictObj.Add(1, objList);
}
objList.Add(. . .);
```

Zadatak 5. Proizvoljno inicijalizovati dictionary čiji ključ je tipa ***int***, a vrednost tipa ***liste stringova***. Iterirati kroz celu kolekciju, i ispisati sve elemente svake liste u kolekciji u sledećem formatu: *<dictionary_key> - <index_of_list_item> - <string_value>*.

KeyValuePair je struktura podataka koja predstavlja key-value par u kolekciji, koju treba iskoristiti u ovom zadatku.

Konvertovanje dictionary kolekcije u listu

Zadatak 6. Inicijalizovati dve proizvoljne dictionary kolekcije istog tipa, i proizvoljno ih inicijalizovati. Kreirati novu dictionary kolekciju koja predstavlja merge ove dve.

Prilikom prolaska kroz dictionary-e potrebno je prvo ih konvertovati u odgovarajuću listu. Konverotvanje dictionary-a u listu podrazumeva kreiranje liste čiji su elementi KeyValuePair strukt ura:

```
Dictionary<TKey, TValue> dictObj = newDictionary<TKey, TValue>(10);
dictObj.ToList() →List<KeyValuePair>
```

Vežba 4 – Napredni koncepti C# programskog jezika

Cilj ove vežbe je da proširi osnovnoznanje programskog jezika C#.

Eksplicitno kastovanje podataka

Eksplicitno kastovanje daje na znanje programskom prevodiocu u kom formatu želimo da tumačimo vrednosti promenljive. Ukoliko ne kastujemo eksplicitno, komapjler će implicitno da kastuje u trenutku kada to bude neophodno. Eksplicitno kastovanje u C# možemo podeliti u dve kategorije 1) *kastovanje korišćenjem prefiksa*, 2) *as kastovanje*. Sintaksa je navedena ispod.

```
Prefix cast:
BaseType g = ...;
SpecificType t = (SpecificType) g;

As cast:
BaseType g = ...;
SpecificType t = g as SpecificType;
```

Uglavnom je praksa da programeri koriste uvek samo jednu od ove dve konstrukcije. Međutim, postoje značajne razlike između ove dve vrste kastovanja koje treba razmotriti pre opredeljenja za konkretan tip kastovanja. Razlike se odnose na ponašanje aplikacije u slučaju neuspešnog pokušaja kastovanja, kao i na performanse.

U slučaju da nije moguće izvršiti kastovanje (*npr. ukoliko SpecificType ne nasleđuje BaseType*) u slučaju prefiks kastovanja biće bačen izuzetak. As kastovanje će vratiti null i nastaviti regularno sa izvršavanjem programa, a problem će se manifestovati tek prilikom korišćenja promenljive, npr. u okviru druge metode ili klase. Ovakve greške i bugove u kodu je mnogo teže otkriti nego što je to u slučaju da se odmah baci izuzetak i na taj način jasno ukaže šta je problem. S druge strane, as-kastovanje je dosta brža operacija u odnosu na prefix-kastovanje (oko pet puta je brže). Na osnovu ovoga, prefiks kastovanje se još naziva i pouzdano kastovanje, a as kastovanje se naziva brzo kastovanje.

Problem vraćanja vrednosti null u slučaju as-kastovanja može se izbeći proverom da li je vrednost null, ali to svakako nosi i određene probleme, kao što je: 1) činjenica da se na taj način ne vidi da li je problem u kastovanju ili je i originalna promenljiva bila null, 2) prilikom bilo kakve dodatne provere gube se prednosti kada su u pitanju performanse as-kastovanja.

Zadatak 1. Napisati program koji inicijalizuje niz od 30000000 elemenata tipa *object*na sledeći način:

- elementima sa indeksom i | (i%3 == 0) dodeliti vrednost null,
- elementima sa indeksom i | (i%3 == 1) dodeliti proizvoljnu string vrednost,
- elementima sa indeksom i | (i%3 == 2) dodeliti novu instancu tipa *object*.

Proći kroz sve elemente niza, i naći zbir dužina svakog stringa. Proveru da li je element niza string implementirati:

1) koristeći*prefix-cast* elemenata niza. Umesto hvatanja izuzetka u slučaju neuspešnog kastovanja, proveriti da li je element niza string, koristeći **is** operator;
2) koristeći*as-cast*.

Rukovanje izuzecima u C#

Izuzeci su nepredviđene greške koje se dešavaju u toku izvršavanja programa Ukoliko je moguće izuzetke bi uvek trebalo uhvatiti, uz odgovarajuće akcije kako bi se sprečilo defektivno ponašanje i obezbedio robustan program.

Implementiranjem izuzetaka omogućuje se transfer kontrole iz jednog dela programa u drugi. Izuzeci se bacaju pomoću ključne reči **throw** gde se nepredviđena greška desila Kada u nekom delu programa očekujemo izuzetak (jer znamo da taj deo programa može da dovede do defektivnog stanja), pravimo **try-catch** blokove. Ovi blokovi uvek idu u paru, ***try***blok sadrži izvorni kod gde očekujemo izuzetak, dok se ***catch*** blok izvrši samo u slučaju kad se navedeni izuzetak desi. Postoji mogućnost korišćenja više catch blokova u slučaju da očekujemo više tipova izuzetka. Ukoliko je to situacija, blokove treba poređati tako da u njima očekivani izuzeci budu poređani od specijalizovanog prema opštijim izuzecima. Opcioni blok prilikom rukovanja izuzecima je **finally** blok koji će se izvršiti bez obzira da li se izuzetak desio ili ne. U skladu sa tim, ovaj blok uglavnom sadrži naredbe koje se odnose na oslobađanje resursa zauzetih u ovom delu programa, npr. ukoliko je fajl otvoren, potrebno je zatvoriti taj fajl bez obzira da li se izuzetak desio ili ne. Sintaksa try-catch-finaly bloka je data ispod.

```
try
{
//izvorni kod gde očekujemo izuzetak
}
catch (ExceptionName1 e1)
{
// programski blok koji treba izvršiti da se desi izuzetak ExceptionName1
}
…
catch (ExceptionNameN eN)
{
// programski blok koji treba izvršiti da se desi izuzetak ExceptionNameN
}
finally
{
//programski
 }
```

Zadatak 2. Izmeniti zadatak 1. tako da se u slučaju neuspešnog kastovanja obradi odgovarajući izuzetak. Ukoliko je element tipa string, vrednost stringa upisati u fajl. Takođe, nakon prolaska kroz sve elemente niza potrebno je osloboditi zauzete resurse.

System.IO.File je osnovna klasa za rukovanje fajlovima u .NET-u. Za rad sa tekstualnim fajlovima može se iskoristiti i *System.IO.TextWriter* klasa.

Prosleđivanje izuzetka

Ukoliko je potrebno izuzetak bačen u jednoj metodi proslediti u drugu metodu (koja je tu metodu pozvala) veoma je važno ispravno rukovati izuzetkom, jer u suprotnom stack trace originalnog izuzetka može biti uništen. Ovakav primer rukovanja izuzetkom nije ispravan.

```
try
{  // deo koda koji može uzrokovati bacanje izuzetka}
catch (Exception e)
{
throw e;   //uzrokovaće da stack trace originalnog izuzetka bude uništen.
}
```

Umesto toga, originalni izuzetak treba proslediti na sledeći način:

```
try
{  // deo koda koji može uzrokovati bacanje izuzetka }
catch (Exception e)
{
// dodatno, praksa lepog programiranja je da se informacija o izuzetku loguje iako se sam
izuzetak obrađuje u drugom delu programa.
throw;
}
```

Takođe, moguće je prepakovati originalni izuzetak kako bi se prilagodio metodi kojoj se prosleđuje, ali ponovo treba voditi računa da se prilikom prepakivanja ne uništi stack trace. Primer ispod prikazuje kako se uhvaćeni Exception1 može prepakovati u drugi Exception2 bez gubljenja stack trace.

```
try
{  // deo koda koji može uzrokovati bacanje izuzetka }
catch (Exception1e)
{
// izuzetak se prepakuje tako da se sačuva stack trace originalnog izuzetka
thrownew Exception2(“”My custom exception message”, e);
}
```

Izbegavanje bacanja suvišnih izuzetaka

Generalno, bacanje izuzetaka je skupa operacija i kad god je moguće, efikasnije je koristiti različite validacije u kodu umesto bacanja izuzetka.

- Iz prethodnog primera, korišćenje *is* operatora je efikasnije nego rukovanje izuzetkom u slučaju neuspešnog kastovanja.
- Na primeru programa koji deli dva broja, umesto korišćenja try-catch bloka kao zaštite od deljenja nulom, mnogo je efikasnije proveriti da li je delilac nula, i u zavisnosti od toga nastaviti sa izvršavanjem koda.

Using blok

Using blok je ekvivalentan **try-finally** bloku, pri čemu try blok ima isti telo kao using blok, dok se u finally delu poziva *Dispose()* metoda objekta koji se zada kao parameter using bloka. Iz tog razloga, da bi se objekat koristio kao parameter using bloka, mora se implementirati interfejs *IDisposable*. Sintaksa using bloka je navdena ispod.

```
using (Resurs r = new Resurs())
{
// operacije nad resursom r
}
```

Zadatak 3. Koristeći using blok, implementirati kreiranje tekstualnog fajla na disku.

Vežba 5 – Korištenje XML-a u C#-u

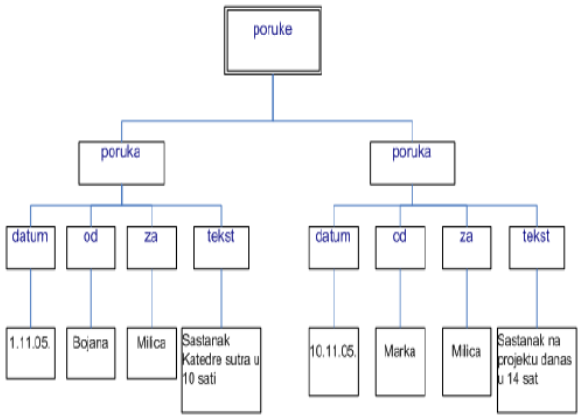
Cilj ove vežbe je da pruži osnovno razumevanje XML jezika kao i njegovo primenu u okviru programskog jezika C#.

XML

XML je standardni skup pravila za definisanje formata podataka u elektronskoj formi. Sledeći pravila XML standarda, korisnici definišu sopstvene (XML) formate podataka, koje mogu koristiti za njihovo skladištenje, obradu i razmenu.

XML je skraćenica za *Extensible Markup Language*, odnosno proširivi (meta) jezik za označavanje (engl. *markup*) tekstualnih dokumenata. Ideja je bila da se stvori jezik koji će i ljudi i računarski programi moći jednostavno da čitaju. XML definiše opštu sintaksu za označavanje podataka pomoću odgovarajućih etiketa (engl. *tags*) koje imaju poznato ili lako razumljivo značenje. Format koji obezbeđuje XML za računarske elemente može se prilagoditi najrazličitijim oblastima, kao što su elektronska razmena podataka, čuvanje podataka, odvajanje podataka od prezentacije, vektorska grafika, sistemi glasovne pošte, izrada novih specijalizovanih jezika za označavanje. XML dokumenti su osetljivi na veličinu slova (engl. *case sensitive*).

```
<?xml version="1.0" encoding="UTF-8"?>
<poruke>
  <poruka>
    <datum>1.11.05. </datum>
      <od>Bojana</od>
       <za>Milicu</za>
       <tekst>Sastanak Katedre sutra u 10 sati </tekst>
  </poruka>
  <poruka>
    <datum>10.11.05. </datum>
      <od>Marka</od>
       <za>Milicu</za>
       <tekst>Sastanak na projektu danas u 14 sati  </tekst>
  </poruka>
</poruke>
```



Slika 1. – Hijerarhijska struktura XML dokumenta

Struktura XML datoteke

Deklaracija

Svaki XML dokument mora započeti sa sledećom linijom kako bi XML interpreteri znali na koji način sadržaj dokumenta treba da se interpretira:

```
<?xmlversion="1.0"encoding="utf-8" ?>
```

Version označava verziju XML standarda koji se koristi u dokumentu, dok *encoding* označava enkodovanje karaktera u dokumentu (UFT-8, UTF-16).

XML elementi

XML datoteka se sastoji od XML elemenata. Elementi predstavljaju par tagova sa sadržajem, što je prikazano u primjeru ispod.

```
<pozdrav> Hello, world! </pozdrav>
```

Elementi se mogu ugnježdavati s tim da se uvek mora zatvoriti element koji je poslednji otvoren.

Validan primer:

```
<message>
  <text>
    Hello, world!
  </text>
</message>
```

Nevalidan primer:

```
<message>
  <text>
   Hello, world!
  </message>
</text>
```

Svaka XML datoteka mora imati isključivo jedan korenski (engl. *root*) element koji sledi neposredno posle zaglavlja. Svi ostali elementi moraju da se nalaze unutar korenskog elementa.

XML Atributi

Svaki XML element može imati nula, jedan ili više atributa koji označavaju dodatne osobine elementa. Atributi se definišu unutar XML elemenata i to uvek bez navodnika, dok vrednosti atributa se uvek navode između navodnike.

```
<message>
  <text size ="12">
    Hello, world!
  </text>
</message>
```

Podaci se mogu skladištiti ili u elementima ili u atributima, kao što je pokazano u naredna dva primera. U prvom primeru *tip* je atribut, dok je u drugom primeru *tip* element. Oba primera daju iste informacije. Ne postoje određena pravila kada koristiti atribute a kada elemente. Preporuka je da se elementi koriste kada je u pitanju nešto što je samo po sebi celokupna informacija, a ne neki njen pomoćni deo.

```
<pozicija tip="nabavljac">
<ime>Pera</ime>
<prezime>Peric</prezime>
```

```
<pozicija>nabavljac</pozicija>
<ime>Pera</ime>
<prezime>Peric</prezime>
```

Komentari

U XML dokumente je moguće umetati komentare kao specijalne vrste elemenata koji će biti ignorisani prilikom parsiranja. Komentari mogu da služe kao dodatno pojašnjenje. XML komentari počinju sa <!-- i završavaju se sa --> karakterima

```
<message>
<!-- Message text begins here -->
<text>
   Hello, world!
</text>
</message>
```

Parsiranje XML datoteke

Iako su XML datoteke čitljive za čoveka, najčešće se kreiraju i čitaju programski. Programski jezik C# ima ugrađene klase za pisanje i čitanje XML datoteka čije korišćenje će biti demonstrirano u ovom poglavlju. Klase za rukovanje XML datotekama se nalaze u *System.Xml* namespace.

Parsiranje

XmlWriter

XmlWriter je klasa koja omogućuje upisivanje objekata iz memorije u XML datoteku. Pre svega, potrebno je napraviti novu instancu ove klase pomoću statičke metode *Create* koja će kreirati prazan xml dokument.

```
using (XmlWriter writer = XmlWriter.Create("students.xml"))
{
        writer.WriteStartDocument();
        writer.WriteEndDocument();
}
```

Nakon toga se kreira korenski element bez kojeg XML nije validan. Treba voditi računa da svaki element koji otvaramo trebamo i zatvoriti. U prethodni isečak koda treba dodati sledeće linije:

```
writer.WriteStartElement("Students");      // <-- Important root element
writer.WriteEndElement();                  // <-- Closes it
```

Ako želimo da upišemo element koji nema ugnježdenog elementa već samo sadržaj prostog tipa možemo uraditi metodom *WriteElementString*.

```
writer.WriteElementString("Index", student.index);
```

Zadatak 1. U xml datoteku "students.xml" upisati tri studenta koji sadrže elemente ime i indeks. Treba upisati sebe, kolegu/koleginicu sa leve i sa desne strane po rasporedu sedenja u laboratoriji, sortirano po broju indeksa.

XmlReader

Ova klasa daje mogućnost da se otvara i učitava XML datoteka. Pruža mogućnost rukovanja xml datotekama na najnižem nivou što uvodi kompleksnost u programe koji koriste ovo rešenje. Omogućuje isključivo kretanje unapred po datoteci (*forward-only*) što smanjuje potrebnu memoriju potrebnu za korišćenje ove klase i povećava performanse iste.

```
using (XmlReader reader = XmlReader.Create("students.xml"))
{
        while (reader.Read())
        {

        }
}
```

Zadatak 2. Otvoriti datoteku kreiranu u prethodnom zadatku, proći kroz sve elemente i ispisati ime studenta ako se broj indeksa studenta poklapa sa svojim brojem indeksa.

Serijalizacija podataka

Serijalizacija je proces konvertovanja objekata u tok bajtova kako bi se isti smeštali u memoriju, bazu podataka, datoteku ili slali preko mreže. Glavni cilj serializacije jeste sačuvanje stanja objekta kako bi se kasnije isti mogao ponovo kreirati. Suprotna operacija od serializacije se zove deserializacija.

Kada se objekti serijalizuju u tok, ne zapisuju se samo podaci iz objekta već i informacija o tipu objekta, verziji objekta, lokalizaciji, biblioteci u kojoj se nalazi itd.

Tokovi implementirani u .NET platformi imaju već ugrađene klase koje služe za serializaciju (i deserializaciju) objekata. Ugrađeni serializatori su binarni, XML i SOAP, ali je moguće napraviti

korisničke klase za serializaciju. Ovaj kurs pokriva i objašnjava samo XML serializaciju jer je njihov izlaz čitljiv za čoveka (i pogodan u edukativne svrhe) i jer čini osnovu SOAP (Simple Object Access Protocol) serializacije na čemu WCF razmena podataka zasniva.

Zadatak 3. Napraviti novu konzolnu aplikaciju "MySerializer", dodati javnu klasu Student koja ima sledeće javne property-je: Name, FamilyName, Index, Birthday. Kreirati i serializovati instancu klase Student pomoću XmlSerializer klase u datoteku "student.xml". Otvoriti kreiranu datoteku i prodiskutovati sadržaj.

Smernice:

- Napraviti klasu Student

```
publicclassStudent
{
   Public string Name { get; set; }
   Public string FamilyName { get; set; }
   Public int Index { get; set; }
   Public DateTime Birthday { get; set; }
}
```

- Napraviti instancu klase *XmlSerializer* koji serializuje objekte tipa Student (potrebno je uključiti namespace System.Xml.Serialization)
- Koristeći instancu klase *TextWriter* (namespace System.IO), pomoću objekta za serializaciju, serializovati instancu klase tipa Student.

```
XmlSerializer serializer = new XmlSerializer(typeof(Student));
using (TextWriter textWriter = new StreamWriter("student.xml"))
{
   serializer.Serialize(textWriter, student);
}
```

Izlazna datoteka sadrži sve XML elemente koji opisuju klasu i vrednost svakog elementa je vrednost koja je upisana u serializovani objekat u programu.

Klasa XmlSerializer podrazumevano sve javne (public) članove i propery-je prosleđene klase pretvori u elemente u odgovarajućoj relaciji. Serializacija ne sadrži informacije o tipu klase. Klasa da bi se serializovala, mora imati podrazumevani konstruktor. Članovi koji su ReadOnly se ne serializuju.

Kako bi se kontrolisalo kako se serializuju članovi (i property-ji) klase, ciljani članovi se mogu dekorisati sledećim atributima:

- *XmlAttribute*: Označava da ce član klase biti atribut u XML datoteci

- *XmlElement*: Označava da će član klase biti XML element (ovo je podrazumevano)
- *XmlIgnore*: Član klase se neće razmatrati prilikom serializacije
- *XmlRoot*: Označava da će element biti korenski element XML datoteke

Zadatak 4. Dodati odgovarajuće atribute u klasu Student iz prethodnog primera, da se Birthday ne serializuje i da broj indeksa studenta bude atribut u XML zapisu iste klase.

Zadatak 5. Deserializovati i spisati sadržaj XML datoteke iz prethodnog zadatka u novu instancu klase Student.

Smernice:

- Napraviti instancu klase *XmlSerializer* koji serializuje objekte tipa Student
- Koristeći instancu klase *TextReader*, pomoću objekta za serializaciju, deserializovati instancu klase tipa Student u novokreiranu i neinicializovanu promenljivu.

```
Student xmlStudent;
XmlSerializer deserializer = new XmlSerializer(typeof(Student));
using (TextReader reader = new StreamReader("student.xml"))
{
   object obj = deserializer.Deserialize(reader);
   xmlStudent = (Student)obj;
}
```

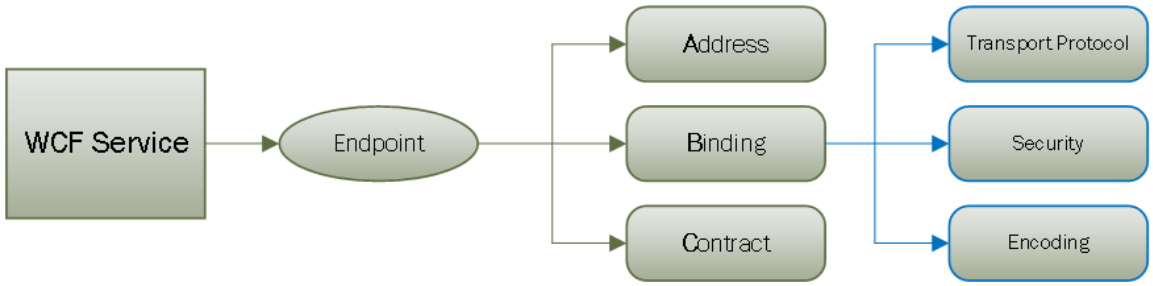
Vežba 6 – Windows Communication Foundation (WCF) (1)

Zbog sve većeg obima programskih rešenja i male međusobne povezanosti među pojedinim komponentama, kao osnovni koncept razvoja velikih, distribuiranih sistema se uvodi arhitektura bazirana na uslugama tj. *service-oriented architecture (SOA)*. U ovakvim programskim rešenjima usluge (*services*) predstavljaju spregu za komunikaciju među pojedinim modulima.Komunikacija se obavlja preko strogo definisanih, strukturiranih poruka za razmenu podataka. Ove poruke (tzv. SOAP poruke)se sastoje iz tela (*body*) u okviru koga se prenose korisnički podaciizmeđu programskih komponenti, i zaglavlje (*header*) koje je namenjeno za prenošenje specijalnih podataka.

Windows Communication Foundation (WCF) je deo .NET razvojnog okruženja za pisanje distirbuiranih, servisno-orijentisanih aplikacija. WCF komponenta se sastoji iz tri osnovne komponente:

1. Definicija usluge, odnosno *Service Contract* u okviru koga se definišu metode koje se mogu pozivati, ili polja koja se mogu koristiti;
2. Aplikacija koja omogućuje pristup uslugama, odnosno servisna aplikacija;
3. Jedna ili više krajnjih tačaka (*endpoint*) preko kojih se pristupa usluzi. Aplikacije koje koriste usluge se nazivaju klijentske aplikacije.

Dakle, servisna aplikacija (WCF servis) je klijentskim aplikacijama (WCF klijenti) dostupna preko krajnjih tačaka, tj. *endpoint*-a. Endpoint predstavlja skup svih neophodnih informacija koje su klijentu potrebne za pristup servisu. Svaki endpoint mora da sadrži informacije o adresi na kojoj servis sluša (***address***), o komunikacionom protokolu (***binding***), kao i informacije o kontraktu (***contract***) u okviru koga se definišuusluge koje servis nudi klijentima. Osnovni elementi endpointa su prikazani na slici 1.



Slika 4. Struktura WCF Endpoint -a

Adresa. Predstavlja adresu na kojoj servis sluša dolazne zahteve. Predstavlja se u standardnom URI formatu, na sledeći način: *Scheme***://**<*MachineName*>*[***:***Port]/Path*, gde je:

- Scheme: transportni protokol (TCP, HTTP, itd.),
- Machine name: ime mačine gde je hostovan servis,

- Port: ovaj parametar je opcioni, i ukoliko se ne navede koristi se default broj porta za odgovarajući protokol.
- Path: putanja do servisa.

Binding. Definiše način komunikacije između klijenta i servisa (komunikacioni protokol). Osim specificiranja mrežnog protokola, binding-om se mogu specificirati i druge karakteristike za komunikaciju kao što su bezbednost i kodiranje koje određuje format poruke - binarni ili tekstualni.

Contract. Definiše usluge koje određeni WCF servis izlaže (*expose*). Klasi koju želimo da izložimo dodajemo atribut *[ServiceContract]*. Ovako nastala klasa se naziva ugovor. Svakoj metodi koja se izlaže kroz ugovor dodaje se atribut *[OperationContract]*. Ukoliko je potrebno da iodređeni složeni podatak bude na raspolaganju klijentu, obeležava se atributom *[DataContract]*, a njegovi članovi prostog tipa *[DataMember]*.

Implementacija WCF servisa i klijenta

Zadatak 1. Implementirati WCFservis koji nudi metode za dodavanje i uklanjanje osoba (definisanih uz pomoć jmbg, punog imena), kao i ispis podataka o svakoj osobi.

Prvi korak prilikom implementacije je uvek definisanje *ServiceContract*-a koji predstavlja funkcionalnosti koje servis nudi. Praksa lepog programiranja je da se kontrakti čuvaju u okviru posebne bibilioteke (*Visual C# Class Library, tj. DLL file*) koja će biti referencirana od strane WCF servis i klijent aplikacije.

```
// Definicija service contract-a
interfaceIWCFService;
```

Sledeći korak je pisanje WCF servis aplikacije. Servisna aplikacija treba da sadrži implementaciju service contract-a, kao i logiku za hostovanje servisa. **ServiceHost** (*System.ServiceModel* namespace) je klasa za hostovanje servisa. Hostovanje servisa podrazumeva: 1) instanciranje novog *ServiceHost* objekta, 2) dodavanje endpointa (*AddServiceEndpoint*).

Za implementaciju WCF servisa koristiće se Visual C# konzolna aplikacija kroz koju će servis

biti startovan/zaustavljen.

```
ServiceHost svc =newServiceHost(typeof(WCFService));
svc.AddServiceEndpoint(typeof(IWCFService),
                       newNetTcpBinding(),
                       newUri("net.tcp://localhost:4000/IWCFService"));
```

Zadatak 2. Implementirati WCF klijent aplikaciju koja se povezuje na servis iz prethodnog zadatka, i nudi mogućnost dodavanja novih osoba, ispisa podataka, kao i uklanjanja postojećih.

Slično kao za implementaciju servisa, i za implementaciju WCF klijenta koristiće se Visual C# konzolna aplikacija kroz koju će biti uspostavljen kanal (*proxy*) prema servisu. .NET nudi dve klase za implementaciju proxy-a ka servisu: *ClientBase* i *ChannelFactory*. U ovom zadatku biće prikazan način korišćenja *ChannelFactory* klase. Kreiranje kanala koristeći *ChannelFactory* navede je ispod. Takođe, i u okviru klijentske aplikacije potrebno je referencirati sistemsku biblioteku **System.ServiceModel**.

```
ChannelFactory< IWCFService> factory = newChannelFactory< IWCFService>(
newNetTcpBinding(),
newEndpointAddress("net.tcp://localhost:4000/IWCFService"));

IWCFService proxy = factory.CreateChannel();
```

Vežba 7 – Windows Communication Foundation (WCF) (2)

Prenošenje složenih podataka preko WCF-a

WCF koristi takozvanu ***DataContract*** serializaciju prilikom prenosa podataka. Svi ugrađeni .NET primitivni tipovi, kao i neki ugrađeni izvedeni tipovi mogu biti serializovani i deserijalizovani pomoću *DataContract serializator-a*.

Svi novi (korisnički) izvedeni tipovi (klase) moraju imati definisane specijalne atribute kako bi ih *DataContract* serializator mogao serijalizovati/deserijalizovati prilikom prenosa. Očekuje se da se serijalizuju svi javni tipovi, javna polja i property-i u novo-izvedenim tipovima. U cilju serijalizacije, članovi klase se mogu dekorisati sledećim atributima iz **System.Runtime.Serialization** namespace-a:

- ***DataContract***: eksplicitno definiše novi tip podatka za serijalizaciju, koristi se za dekorisanje klasa, struktura i enumeracija;
- ***DataMember***: definiše članove klase koji će biti uključeni u proces serijalizacije;
- ***IgnoreDataMember***: navodi se kako bi se preskočila serializacija određenog člana.

Zadatak 1. Izmeniti metode IWCFService kontrakta tako se umesto parametara jmbg, firstName, lastName koristi objekat klase *Person* sa odgovarajućim poljima. Prilagoditi WCF servis i klijent aplikaciju ovim izmenama.

Rad sa WCF izuzecima

WCF podržava propagaciju grešaka od servisa do klijenata preko izuzetaka. Rad sa izuzecima se omogućava usledeća tri koraka:

1. Metoda interfejsa se definiše da može da baci određeni tip WCF izuzetka. Za ovo je potrebno definisati klasu koja će sadržati opis greške.
2. Servis baci WCF izuzetak
3. Klijent hvata WCF izuzetak

Označavanje WCF metode za bacanje izuzetka se vrši korišćenjem atributa *FaultContract* na sledeći način:

```
[ServiceContract]
public interface WCFInterface
{
    [OperationContract]
    [FaultContract(typeof(MyException))]
    void AddPerson(Person person);
}
```

Prilikom bacanja WCF izuzetka, inicijalizuje se objekat klase koja predstavlja izuzetak, i izuzetak se baca kao što je navedeno u primeru ispod:

```
MyException ex = new MyException();
ex.Reason = "Jmbg already exist.";
throw new FaultException<MyException>(ex);
```

Hvatanje fault exception-a je prikazano u primeru ispod:

```
catch (FaultException<MyException> ex)
{
  Console.WriteLine("Error: " + ex.Detail.Reason);
}
```

Zadatak 2. Implementirati klasu *MyException* sa poljem *reason*, uključujući i odgovarajući property. Implementirati metode AddPerson i RemovePerson tako da se baci odgovarajući izuzetak ukoliko se pokuša dodati osoba sa postojećim jmbg-om, odnosno ukoliko se pokuša ukloniti nepostojeća osoba. Obezbediti odgovarajuće rukovanje izuzetkom.

Konfigurisanje WCF servisa i klijenta iz konfiguracionih fajlova

Platforma .NET podržava konfigurisanje izvršnih aplikacija pomoću konfiguracione datoteke koja sadrži konfiguraciju definisanu na ***eXtensible Markup Language (XML)*** jeziku. Prevođenjem projekta se pravi kopija ove datoteke sa promenjenim imenom ***ime_aplikacije.exe.config***. Menjanje konfiguracije koja se nalazi u ovim datotekama menja podešavanja programa bez potrebe za rekonfigurisanjem.

Ovakve datoteke se dodaju u projekte koje su izvršnog tipa (konzolne i windows aplikacije) na sledeći način: Desni klik na ciljani projekat → Add New Item → Odabrati ”Application Configuration File”. Za rad sa konfiguracionim datotekama potrebno je uključiti **System.Configuration** namespace.

Konfigurisanje WCF servisa

Da bi se servis mogao kreirati samo pomoću imena, bez programskog dodavanja endpoint-a, potrebno je definisati konfiguracionu datoteku servisa na sledeći način:

```
<system.serviceModel>
  <services>
    <service name="WCFService.Service">
     <host>
          <baseAddresses>
          <add baseAddress="net.tcp://localhost:4000/IWCFInterface" />
          </baseAddresses>
     </host>
     <!-- Service Endpoints -->
        <endpoint address=""binding="netTcpBinding"contract="WCFCommon.IWCFInterface" />
    </service>
  </services>
</system.serviceModel>
```

Konfigurisanje WCF klijenta

Ukoliko u klijentsku aplikacionu datoteku dodamo konfiguraciju navedenu ispod, klijent će moći da dobavi sva podešavanja iz konfiguracione datoteke kako bi uspostavio vezu:

```
<system.serviceModel>
  <client>
    <endpoint name="WCFCommon.IWCFInterface"
          address="net.tcp://localhost:4000/IWCFInterface"
          binding="netTcpBinding"
          contract="WCFCommon.IWCFInterface" />
  </client>
</system.serviceModel>
```

Zadatak 3. WCF servisnu i klijentsku aplikaciju iz prethodnog zadatka izmeniti tako da se koriste konfiguracioni fajlovi.

Zadatak 4. Implementirati kalkulator kompleksnih brojeva koji podržava četiri osnovne operacije nad kompleksnim brojevima. WCF servis nudi usluge izvršavanja operacija kalkulatora, dok klijenti unose kompleksne brojeve i operaciju, i očekuju rezultat. U slučaju da klijent pokuša da deli s nulom server treba da baci izuzetak koji klijent treba da uhvati i ispiše.

Vežba 8 – Mehanizmi bezbednosti

Data je implementacija WCF klijenta i servisa za zapisivanje i čitanje podataka na server. Cilj ovih vežbi, da se uvedu bezbednosni mehanizmi za autentifikaciju i autorizaciju klijenata na server prilikom pristupa podacima.

Autentifikacija

Iako .NET i WCF sadrže bogat skup ugrađenih bezbednosnih mehanizama, ovde će u edukativne svrhe biti prikazan jednostavna realizacija autentifikacije koja ne koristi te mehanizme.

Ovaj mehanizam autentifikacije, tj. provere identiteta entiteta koji želi da komunicira, će se realizovati u sledećim koracima:

1) Definisanje klase koja opisuje korisnika
2) Dodavanje kontejnera korisnika
3) Definisanje korisnika
4) Dodavanje metode za autentifikaciju
5) Pamćenje autentifikovanog korisnika

Klasa za definisanje opisa korisnika u distribuiranom sistemu je data u sledećem listingu:

```
publicclassMyUser
{
   public MyUser(string user, string pass)
   {
      _username = user;
      _password = pass;
   }

   publicstring Username
   {
      get { return _username; }
   }
   publicstring Password
   {
      get { return _password; }
   }

   public bool Authenticated
  {
     get { return _authenticated; }
     set { _authenticated = value; }
  }

   bool _authenticated;
  string _username;
  string _password;
}
```

Opisi korisnika mogu da se čuvaju u nekom od standardnih .NET kontejnera. Najčešće će to biti Dictionary, koji iako nije optimizovan za malo zauzeće memorije, omogućava brzu pretragu po ključu.U produkcionom sistemu bi se skladištili u nekoj bazi podataka (LDAP, Active Directory, ili sl.)

Dodavanje kontejnera korisnika na server i definisanje dva korisnika tipa MyUser je prikazano u sledećem listingu:

```
public static readonly Dictionary<string, MyUser> Users=new Dictionary<string, MyUser>();

public ServerDatabase()
{
  Console.WriteLine("Adding users to server...");

  MyUser pera = new MyUser("pera", "P3rA");
  ServerDatabase.Users.Add(pera.Username, pera);

  MyUser admin = new MyUser("admin", "pr3Axadmin");
  ServerDatabase.Users.Add(admin.Username, admin);
}
```

Provera identiteta entiteta se nakon definisanja korisnika proverava u metodi Authenticate na sledeći način:

```
publicstring Authenticate(string user, string pass)
{
  Console.WriteLine("Authenticaticating...");
  if (ServerDatabase.Users.ContainsKey(user))
  {
    If (ServerDatabase.Users[user].Password == pass)
    {
      ServerDatabase.Users[user].Authenticated = true;
      return"Success"; // value from dictionary
    }
    else
    {
       SecurityException ex = newSecurityException();
       ex.Reason = "Invalid password.";
       thrownewFaultException<SecurityException>(ex);
    }
  }
  else
  {
   SecurityException ex = newSecurityException();
   ex.Reason = "Invalid username.";
   thrownewFaultException<SecurityException>(ex);
 }
}
```

Metoda Authenticate iz gornjeg listinga baca WCF izuzetak ako je pozvana sa nepostojećim korisničkim imenom (prvi if), ili ako se uz postojeće korisničko ime prosledi neispravna lozinka (unutrašnji if). U oba slučaja se baca WCF izuzetak i detalju se nalazi primerak klase SecurityException.

Klijent regularno poziva metodu Authenticate posle ovih izmena, ali mora da vodi računa o tome da ta metoda može da baci WCF izuzetak.

Kako bi koristili podršku autentifikacije u drugim servisima, treba u svim metodama gde je potrebno dodati proveru da li je korisnik autentifikovan. Ovo podrazumeva proširivanje potpisa metoda sa korisničkim imenom za kojeg se proverava identitet i zatim proveravanje da li je korisnik autentifikovan. Primer koda za proveru da li je korisnik autentifikovan se nalazi u listingu ispod:

```
public static bool IsUserAuthenticated(string username)
{
  if (ServerDatabase.Users.ContainsKey(username))
  {
    return ServerDatabase.Users[username].Authenticated;
  }
  else
  {
    return false;
  }
}
```

Autorizacija

Autorizacija je bezbednosni mehanizam koji se bavi dodelom i proverom prava pristupa. Poput autentifikaci-je, ovde ćemo (u edukativne svrhe) ručno implementirati ono što je inače ugrađeno u .NET Framework i u WCF.

Autorizaciju ćemo implementirati u sledećim koracima:

1) Definicija prava pristupa
2) Omogućavanje pamćenja prava pristupa u opisu korisnika – tzv. Access Control List (ACL) – spisak prava
3) Dodela prava pristupa postojećim korisnicima
4) Dodavanje metode koja proverava da li korisnik ima pravo da vrši neku akciju
5) Poziv metode za proveru prava pristupa gde god je to potrebno

Prava pristupa ćemo definisati u obliku enumeracije koja sadrži sva moguća prava određene klase. Sledeći listing sadrži pojedinačna prava za pristup svim metodama svih interfejsa (neki od interfejsa će biti dodati na narednim vežbama).

```
publicenumERights
{
   Read, Write
}
```

Klasa za opis korisnika MyUser se proširuje listom prava pristupa u obliku kontejnera u koji se stavljaju pojedinačna prava koja korisnik ima, sa metodom za dodavanje prava pristupa (AddRight) i sa metodom za proveru da li korisnik ima neko pravo (HasRight)

```
HashSet<ERights> _rights = new HashSet<ERights>();

public void AddRight(ERights right)
{
  if (!_rights.Contains(right))
  {
    _rights.Add(right);
   }
}

public bool HasRight(ERights right)
{
  return _rights.Contains(right);
}
```

Dodela prava se vrši prilikom kreiranja korisnika. U produkcionom sistemu bi se ova prava skladištila na repozitorijumu tipa LDAP, Active Directory ili sl. Sledeći listing pokazuje kako možemo da dodelimo različita prava ranije definisanim korisnicima.

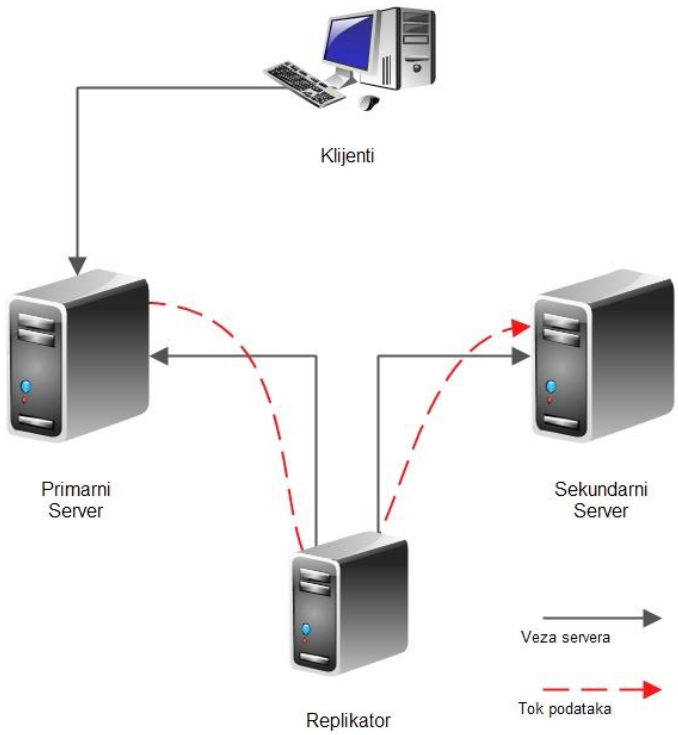
U svim metodama koje korisnik može pozvati, potrebno je dodati autorizaciju kako bi se podaci i operacije servera zaštitili od neautorizovanih korisnika.

```
public static bool IsUserAuthorized(string username, ERights right)
{
   if (ServerDatabase.Users.ContainsKey(username))
  {
    return ServerDatabase.Users[username].HasRight(right);
  }
  else
  {
    return false;
   }
}
```

Vežba 9 – Replikacija

Replikacija se uvodi u distribuirane sisteme radi poboljšavanja preformansi sistema i povećavanja dostupnosti istog. Jedan je od osnovnih elemenata distribuiranih sistema.

Da bi se implementira replikacija, potrebno je „prostu“ klijent – server arhitekturu distribuiranog sistema proširiti. Uvodi se dodatni server (dodatni serveri) koji služi kao sigurnosna kopija (backup ili secondary) server, koji će preuzeti ulogu primarnog servera po potrebi ili u slučaju da dođe do otkaza istog. Dodatno se uvodi servis (unutar svakog od servera) koji replicira podatke sa primarnog servera na sekundarni.



Osnovna zadatak replikatora je kopiranje svih bitnih podataka sa jednog servera na drugi server. Što su podaci bitniji, to se češće repliciraju. U distribuiranim sistemuma sa kritičnom misijom se svaka promena nad podacima replicira.

Kako bi se implementirala replikacija, potrebno je uraditi sledeće korake:

1. U server dodati WCF klijent koji će se povezati sa drugim serverom
2. U server dodati podrsku da se svaki upisani podatak smeštanja lokalno, „replicira“ preko novododate klijentske veze na drugi server

Dodati WCF klijenta u *Main* metodu *WCFServer*dodati metodu kojom se poveyujemo sa drugim serverom na koji repliciramo dospele podatke:

```
private static void ConnectToRemoteServer()
{
do
  {
    try
     {
         ChannelFactory<IDataAccessServer> dataFactory =
newChannelFactory<IDataAccessServer>(typeof(IDataAccessServer).ToString());
         ChannelFactory<IUserServer> userFactory =
newChannelFactory<IUserServer>(typeof(IUserServer).ToString());

         IUserServer replicatorUserService = userFactory.CreateChannel();
         ServerDatabase.ReplicatorDataService = dataFactory.CreateChannel();

        replicatorUserService.Authenticate("replicator", "repa");

        ServerDatabase.IsReplicatorConnected = true;
     }
     catch
     {
         ServerDatabase.IsReplicatorConnected = false;
     }

    Thread.Sleep(1000);
  } while (!ServerDatabase.IsReplicatorConnected);
}
```

Pored koda za povezivanje sa drugim serverom, trebamo dodati i klijentski deo WCF konfiguracije u applokacionu konfiguracionu datoteku. Voditi računa o tome, server izlaže servise na portu 4000, a preko klijentske WCF sprege se sa drugim serverom povezuje na portu 5000.

Da bi se svaki dospeli podatak na serveru replicirao na drugi server, potrebno je isti poslati ka drugom serveru neposredno nakon zapisivanja u lokalni kontejner. Kako bi se beskonačna cirkularna replikacija izbegla u našem primeru ćemo koristiti korisničko ime.

Pre svega je potrebno dodati na server novog korisnika „replicator“ koji ima prava za upis podataka. Zatim treba za svaku dospelu poruku proveriti da li je dosao od strane ovog korisnika, ukoliko jeste, znaci da se radi o replici i trebamo podatak samo lokalno zapisati. Ukoliko je dosao od nekog drugog korisnika, pored zapisivanja trebamo i replicirati podatak. Ovo radimo tako što dodajemo sledeći isečak koda u *Write* metodu *DataAccessServer* klase neposredno nakon zapisivanja podatka lokalno:

```
if (!username.Equals("replicator"))
{
    ServerDatabase.ReplicatorDataService.Write("replicator", id, text);
}
```

Kako bi smo proverili rad replikacionog mehanizma potrebno je uraditi sledeće:

1. Prevesti sve programe
2. Iskopirati iznazne datoteke iz *WCFServer* na dva mesta, na jednom mestu u aplikacionom konfiguracionom fajlu prepraviti da se servisi pokreću na portu 5000, a da se povezuju sa drugim serverom na portu 4000
3. Pokrenuti obe instance servera
4. U klijentu napraviti beskonačnu petlju za poziv *Write* serverske metode sa slučajnim podacima

Vežba 10 – Otpornost na otkaze

Kao što smo na prethodnim vežbama videli, replikacija omogućuje da imamo kopiju bitnih podataka u distribuiranom sistemu, međutim ukoliko dođe do otkaza jednog od server sistema, klijenti neće imati pristup operacijama i podacima. Kako bi se ovo izbeglo, potrebno je da obezbedimo otpornost na otkaze u distribuiranom sistemu (fault tolerance).

Kako bi sistem u potpunosti bio otporan na otkaze, potrebno je za ovo uvesti podršku kako na serverima tako i na klijentima, tj. svi čvorovi sistema moraju biti svesni stanja sistema.

Treba da uvedemo dodatni WCF ugovor koji će služiti da server međusobno mogu da saznaju u kom je drugi (udaljeni) stanju. Uvodimo interface *ISystemMonitor* sa jednom metodom *CheckIfAlive()*.

Implementacija serverskog dela rešenja

Ovaj interfejs implementira server i to po sledećem listingu:

```
public bool CheckIfAlive()
{
   lock (ServerDatabase.ServerStateSync)
   {
      return ServerDatabase.ServerRunning;
   }
}
```

Vidimo da serverska baza podataka treba da sadrži informaciju koja indikuje da li su korisnički servisi pokrenuti. Pošto se ova vrednost postavlja iz više niti, mora njoj da se pristupa iz kritične sekcije.

Ova promenljiva se postavlja na mestima gde se korisnički servisi na serveru pokreću isto iz kritične sekcije.

```
private static void StartServices()
{
if (!_serversStarted)
   {
      _dataServiceHost = new ServiceHost(typeof(DataAccessServer));
      _userServiceHost = new ServiceHost(typeof(UserServer));

      _dataServiceHost.Open();
      _userServiceHost.Open();

      lock (ServerDatabase.ServerStateSync)
      {
         ServerDatabase.ServerRunning = true;
      }
       Console.WriteLine("SERVER: Ready and waiting for requests.");
      _serversStarted = true;
   }
}

private static void StopServices()
{
   if (_serversStarted)
   {
      _dataServiceHost.Close();
      _userServiceHost.Close();

      lock (ServerDatabase.ServerStateSync)
      {
         ServerDatabase.ServerRunning = false;
      }
      Console.WriteLine("SERVER: Stopped.");
      _serversStarted = false;
   }
}
```

Takođe je potrebno uvesti stanje servera, tj neki indikator da li se server ponaša kao primarni ili sekundarni (hot, standby). Ovo ćemo uraditi pomoću sledeće enumeracije:

```
public enum ServerKind
{
   Unknown,
   Primary,
   Secondary
}
```

Glavna metoda servera više ne treba direktno da pokreće izlaganje izlaganje korisničkih servisa, već da pokrene nit u kojoj se nalazi algoritam za nadgledanje sistema. Ova nit treba prvo da proveri da li postoji drugi server koji izlaže servis za nadgledanje, ako postoji treba da "pita" udaljeni server u kom je stanju. Nakon toga treba da odredi svoje stanje i da sam pokrene

izlaganje servisa za nadgledanje kako bi udaljeni server mogao isti algoritam da izvrši. Ukoliko server pređe u stanje Primarnog servera sistema, treba da pokrene izlaganje korisničkih servisa.

```
private static void SystemMonitorThread()
{
   ServerKind serverKind = ServerKind.Unknown;

  _systemMonitorServiceHost = new ServiceHost(typeof(SystemMonitor));
  _systemMonitorServiceHost.Open();

  while (true)
  {
   try
      {
         _systemMonitorFactory = new ChannelFactory<ISystemMonitor>("RemoteSystemMonitorService");
         _systemMotitor = _systemMonitorFactory.CreateChannel();
         bool isRemoteAlive = _systemMotitor.CheckIfAlive();

         serverKind = isRemoteAlive ? ServerKind.Secondary : ServerKind.Primary;

         //Console.WriteLine("SYSMON: Remote server found, assigning {0} role to this sever.", _serverKind);
      }
      catch (EndpointNotFoundException)
      {
          serverKind = ServerKind.Primary;

//Console.WriteLine("SYSMON: Remote server not found, assigning {0} role to this sever.", _serverKind);
      }
      finally
      {
         if (serverKind != _serverKind)
         {
           _serverKind = serverKind;
           Console.WriteLine("SYSMON: Server role now: {0}", _serverKind);
         }

         if (_serverKind.Equals(ServerKind.Primary))
         {
         StartServices();
         }
         else
         {
         StopServices();
         }
         }

         Thread.Sleep(1000);
         }
}
```

Treba obratiti pažnju da se servis za nadgledanje servera i izlaže i konzumira na istom serveru, treba u skladu stim da se napravi konfiguraciona datoteka servera.

Nakon ove su serveri svesni tuđeg stanja i ukoliko dođe do otkaza jednog servera, drugi će preuzeti ulogu primarnog. Međutim replikator nije svestan stanja servera, te ne zna u kom trenutku je koji server izvor, a koji odredište za replikaciju. Najlakši način da i sam mehanizam replikacije bude svestan stanja jeste da replikacionu logiku uključimo u sam server.

Kako bi se replikacione servisne metode jasno odvojile od korisničkih metoda, izdvajaju se u poseban servisni ugovor. Radi lakše implementacije, u ovom primeru se će se koristiti replikaciona logika bez bezbednosnih mehanizama.

```
[ServiceContract]
public interface IReplicator
{
   [OperationContract]
   void WriteUpdated(Dictionary<long, string> updatedValues);
}
```

Replikaciona logika se sad u potpunosti implementira na serveru. Servisni deo replikacije se implementira na sledeći način:

```
public class Replicator : IReplicator
{
public void WriteUpdated(Dictionary<long, string> updatedValues)
   {
if (updatedValues.Count > 0)
      {
Console.WriteLine("REPLICATION: {0} values replicated.", updatedValues.Count);
      }

foreach (long item in updatedValues.Keys)
      {
ServerDatabase.Data[item] = updatedValues[item];
      }
   }
}
```

Logika koju je replikacioni klijent izvršavao, se sad premešta na server i to u posebnu nit koja će se pokretati nakon niti za nadgledanje servera (System Monitor).

```
private static void ReplicatorThread()
{
   _replicatorServiceHost = new ServiceHost(typeof(Replicator));
   _replicatorServiceHost.Open();

while (true)
   {
try
     {
        _replicatorFactory = new ChannelFactory<IReplicator>("RemoteReplicator");
        _replicator = _replicatorFactory.CreateChannel();

if (_serverKind.Equals(ServerKind.Primary))
        {
int count = 0;
lock (ServerDatabase.ReplicationQueueLock)
           {
              _replicator.WriteUpdated(ServerDatabase.ReplicationQueue);
              count = ServerDatabase.ReplicationQueue.Count;
ServerDatabase.ReplicationQueue.Clear();
           }

if (count > 0)
           {
Console.WriteLine("REPLICATION: Replication of {0} items finished.", count);
           }
        }
else
        {
//Console.WriteLine("REPLICATION: Not replicating from secondary server.");
        }
     }
catch (EndpointNotFoundException)
     {
//Console.WriteLine("REPLICATION: Cannot connect to remote replication service.");
     }

Thread.Sleep(2000);
   }
}
```

Implementacija klijentskog dela rešenja

Nakon implementacije replikatora, serveri su svesni stanja sistema i replikacija na osnovu stanja sistema određuje koji server je izvor, a koji odredište za replicirane podatke. Preostaje prilagođenje klijenta, tj proširenje klijenta tako da ako izgubi vezu sa jednim serverom pokuša da se poveže sa drugim serverom sistema. U ovom primeru se zbog pojednostavljenja implementacije se izbacuje podrška za keširanje.

```
private int nextTry = 0;

private List<ChannelFactory<IUserServer>> _userFactories = new List<ChannelFactory<IUserServer>>();
private IUserServer _userService;
private string _username = string.Empty;
private string _password = string.Empty;

private List<ChannelFactory<IDataAccessServer>> _dataFactories = new List<ChannelFactory<IDataAccessSer
ver>>();
private IDataAccessServer _dataService;

private bool _isClientConnected = false;

public DataAccessServiceClient(string username, string password)
{
   _username = username;
   _password = password;

   _userFactories.Add(new ChannelFactory<IUserServer>("ServerOne.IUserServer"));
   _userFactories.Add(new ChannelFactory<IUserServer>("ServerTwo.IUserServer"));

   _dataFactories.Add(new ChannelFactory<IDataAccessServer>("ServerOne.IDataAccessServer"));
   _dataFactories.Add(new ChannelFactory<IDataAccessServer>("ServerTwo.IDataAccessServer"));
}
```

Klijent pre svega treba da učita adrese svih dostupnih servera kako bi se konektovao na onaj koji je trenutno dostupan. Treba voditi računa o tome da se prilagode nazivi i adrese pristupnih tačaka servera u aplikacionoj konfiguracionoj datoteci klijenta. Takođe treba da se implementiraju metode za povezivanje i raskidanje veze sa serverom:

```
public void Connect()
{
while (!_isClientConnected)
  {
    _userService = _userFactories[nextTry].CreateChannel();
    _dataService = _dataFactories[nextTry].CreateChannel();

try
    {
      _userService.Authenticate(_username, _password);
      _isClientConnected = true;
Console.WriteLine("Client conneced to server at: {0}", _userFactories[nextTry].Endpoint.Name);
    }
catch (EndpointNotFoundException)
    {
Console.WriteLine("Client could not connect to server at: {0}", _userFactories[nextTry].Endpoint.Name);
      nextTry = (nextTry + 1) % 2;
      _isClientConnected = false;
    }
  }
}
```

```
public void Disconnect()
{
  DisconnectService((ICommunicationObject)_dataService);
  DisconnectService((ICommunicationObject)_userService);

  _isClientConnected = false;
Console.WriteLine("Client disconnected from server.");
}

private void DisconnectService(ICommunicationObject service)
{
if (service != null)
  {
try
    {
      service.Close();
    }
catch
    {
if (service != null)
      {
        service.Abort();
      }
    }
  }
}
```

Svaki put kad se desi komunikaciona greška pri pozivanju operacije na serveru, klijent mora da pokuša da se poveže na drugi server sve dok se ne uspostavi veza.

```
public void Write(long id, string text)
{
bool written = false;
do
   {
try
      {
         _dataService.Write(_username, id, text);
         written = true;
      }
catch (Exception)
      {
         _isClientConnected = false;
         Connect();
      }
   } while (!written);
}

public string Read(long id)
{
bool read = false;
string returnValue = null;
do
   {
try
      {
         returnValue = _dataService.Read(_username, id);
         read = true;
      }
catch (Exception)
      {
         _isClientConnected = false;
         Connect();
      }
   } while (!read);
return returnValue;
}
```

Nakon ovakve implementacije pozivanja serverskih metoda, moguće je proizvoljno gašenje servera, sistem će se uvek oporaviti od greške i podaci će uvek biti konzistentni na svim serverima.