

Laboratorijska Vježba 4. Objektno Orijentisani Principi u C#

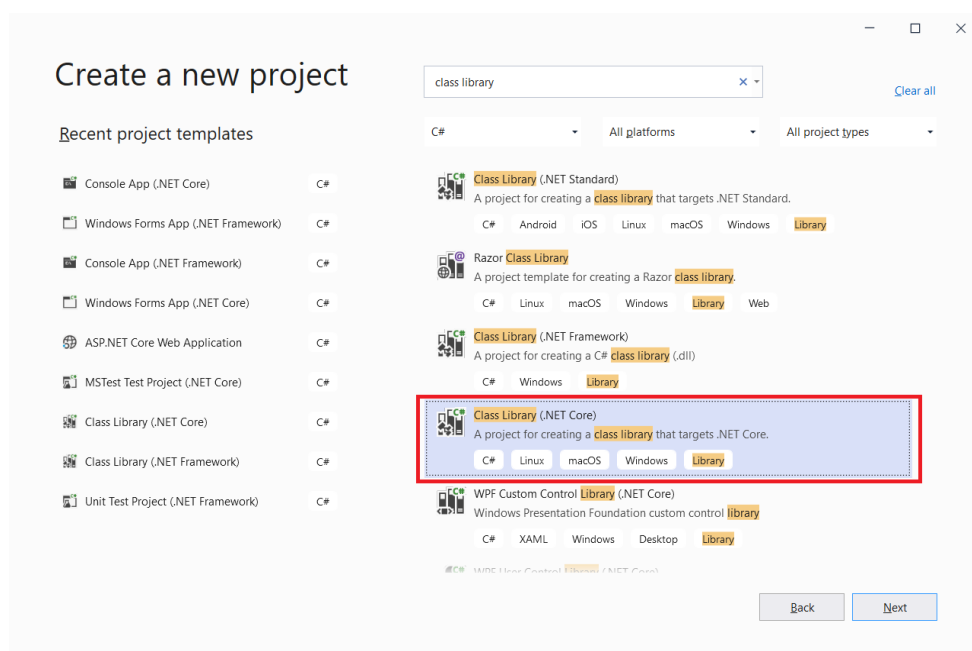
Cilj vježbe:

- Kreiranje DLL biblioteke klasa u *Visual Studio* okruženju;
- Upoznavanje sa klasama i interfejsima u C#;
- Korištenje *properties* metoda i hijerarhije nasljeđivanja klasa.

1. Kreiranje DLL biblioteke klasa u *Visual Studio* okruženju

Konzolne aplikacije pružaju najjednostavniji način interakcije sa korisnikom (prikaz rezultata na ekranu, omogućavanje korisničkog unosa i sl.). Osim konzolnih aplikacija, postoji veliki broj drugih vrsta aplikacija (npr. *Windows Forms* desktop aplikacije, ASP.NET web-aplikacije) koje također omogućavaju interakciju sa korisnikom. Ukoliko se isti programski kod želi koristiti u različitim projektima, nije isplativo taj kod replicirati na više mjesta (posebno ukoliko je potrebno nad njim vršiti neke promjene). Iz tog razloga često se vrši kreiranje DLL biblioteka klasa koje sadrže programski kod definisan na jednom mjestu. Taj kod se zatim može uključiti u druge projekte i koristiti kao da je u pitanju neka od predefinisanih biblioteka (poput *System.Collections* i drugih).

U *Visual Studio* okruženju postoje različite vrste bibliotečnih projekata. Najjednostavnije je izvršiti pretragu za pojam *class library*, a nakon toga odabrati vrstu projekta **Class library (.NET Core)** sa oznakom **C#**, na način prikazan na Slici 1. Nakon toga potrebno je izvršiti konfiguraciju na isti način kao za konzolne aplikacije u prethodnim laboratorijskim vježbama.

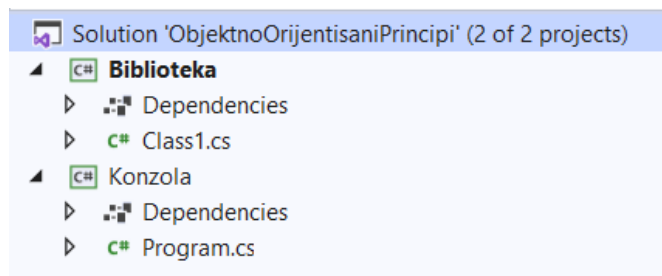


Slika 1. Odabir projekta biblioteke klasa

Kreirana biblioteka ne može se pokretati (jer ne postoji konzola ili neko drugo okruženje za interakciju sa korisnikom). Iz tog razloga u programskom rješenju koje sadrži projekat biblioteke klasa potrebno je definisati još jedan projekat koji će predstavljati konzolnu aplikaciju koja će koristiti novokreiranu biblioteku klasa.

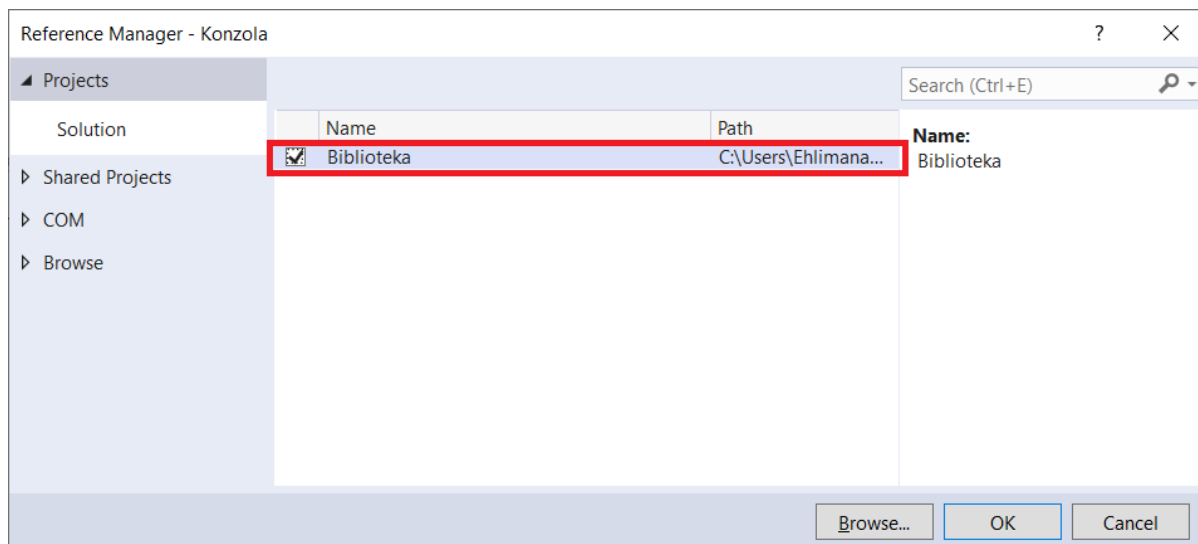
Objektno Orijentisana Analiza i Dizajn

Nakon kreiranja programskog rješenja i oba projekta, dobiva se prikaz kao na Slici 2. Sada je još potrebno uključiti novokreiranu biblioteku u projekat konzolne aplikacije kako bi se biblioteka klasa mogla koristiti bez ikakvih smetnji. U tu svrhu potrebno je izvršiti desni klik na **Dependencies** opciju projekta konzolne aplikacije i odabrati opciju **Add Project Reference**.



Slika 2. Hijerarhija programskog rješenja sa pripadajućim projektima

Kada je projekat biblioteke klasa definisan u istom programskom rješenju kao i projekat koji treba koristiti tu biblioteku, nakon odabira opcije za dodavanje reference na projekat ta biblioteka se odmah prikazuje (kao na Slici 3). Potrebno je samo označiti kvadratić pored biblioteke i potvrditi izbor kako bi biblioteka bila uključena u projekat. Nakon toga moguće je koristiti klase iz projekta koristeći *using* direktive kao i u prethodnim laboratorijskim vježbama. U slučaju kada se željena biblioteka ne nalazi u istom programskom rješenju (ili se posjeduje samo DLL *file* sa definicijom biblioteke), dovoljno je pozicionirati se na lokaciju bibliotečnog *file*-a kako bi se referenca uspješno registrovala.



Slika 3. Dodavanje reference na biblioteku klasa

Sada je moguće početi koristiti biblioteku klasa u konzolnoj aplikaciji. U nastavku će biti objašnjen način kreiranja klasa koristeći objektno orijentisane principe u C# programskom jeziku.

2. Kreiranje sistema klasa u C#

U prethodnim laboratorijskim vježbama korištene su klase iz predefinisanih biblioteka (npr. *Int32*, *Console*, *Tuple*, *List*). Sada je potrebno kreirati sopstvene klase, što će predstavljati osnovu za sve aplikacije koje će se kreirati u okviru narednih laboratorijskih vježbi. Preporučuje se da se sve klase razdvajaju u zasebne *file*-ove zbog lakše navigacije kroz programsko rješenje. U nastavku će rad sa sistemom klasa biti demonstriran na jednostavnom primjeru. Neka je potrebno napraviti aplikaciju koja omogućava rad sa bankom. Banka ima svoju bazu transakcijskih računa, kao i bazu klijenata. Svaki klijent ima svoj identifikacijski broj, ime i jedan ili više transakcijskih računa. Svaki transakcijski račun ima svoj jedinstveni ID broj, kao i informaciju o trenutnom stanju, a limit za podizanje novca s računa u okviru jedne transakcije je 1000 KM. Potrebno je omogućiti sve CRUD (*Create-Read-Update-Delete*) operacije nad klijentima i računima banke.

Prva klasa koja će se definisati je klasa *Klijent*, na način prikazan u isječku koda ispod. Ova klasa označena je kao *public* jer je važno da joj se može pristupiti kroz druge klase (npr. u okviru klase *Banka*). Ona posjeduje tri atributa – identifikacijski broj, ime i listu transakcijskih računa. Veoma je važno omogućiti rad sa atributima instanci klase, odnosno omogućiti da se u nekoj od metoda drugih klasa vrši izmjena vrijednosti ovih atributa. U tu svrhu definišu se **properties**, odnosno *get-and-set* metode za attribute klase. Npr. *property Ime* omogućava da se dobije vrijednost imena nekog klijenta, kao i da se to ime promijeni (kao da su definisane metode *getIme()* i *setIme()*). U slučaju *Identitet property*-a, *set* metoda nije definisana, što znači da se može pristupiti vrijednosti imena, ali da se ta vrijednost ne može promijeniti. Ova vrijednost inicijalno se postavlja u okviru konstruktora koji prima dva parametra.

```
public class Klijent
{
    #region Atributi

    int identitet;
    string ime;
    List<Racun> racuni = new List<Racun>();

    #endregion

    #region Properties

    public int Identitet { get => identitet; }
    public string Ime { get => ime; set => ime = value; }
    public List<Racun> Racuni { get => racuni; set => racuni = value; }

    #endregion

    #region Konstruktor

    public Klijent(int id, string name)
    {
        identitet = id;
        ime = name;
    }

    #endregion
}
```

Objektno Orijentisana Analiza i Dizajn

Sljedeća klasa koju je potrebno definisati je klasa *Racun*. Ona se definiše na sličan način (što je prikazano u isječku koda ispod), uz nekoliko dodataka:

- U ovoj klasi postoji **statički** (*static*) **atribut** koji je isti za sve instance klase.
- Ova klasa osim atributa i *properties* posjeduje i **metode** *PoloziNovac* i *DigniNovac* koje omogućavaju promjenu stanja računa. U njima se vrši provjera da li je vrijednost koja se želi položiti ili povećati pozitivna i u suprotnom dolazi do pojave izuzetka.

```
public class Racun
{
    #region Atributi

    static decimal limitZaPodizanje = 1000;
    decimal stanje;
    int id;

    #endregion

    #region Properties

    public decimal Stanje { get => stanje; }
    public int Id { get => id; set => id = value; }

    #endregion

    #region Konstruktor

    public Racun(decimal pocetnoStanje)
    {
        stanje = pocetnoStanje;
    }

    #endregion

    #region Metode

    public void PoloziNovac(decimal kol)
    {
        if (kol < 0)
            throw new Exception("Količina novca koju polažete na račun mora biti pozitivna.");
        stanje += kol;
    }

    public void DigniNovac(decimal kol)
    {
        if (kol < 0)
            throw new Exception("Količina novca koju podižete sa računa mora biti pozitivna.");
        else if (kol > limitZaPodizanje)
            throw new Exception("Količina novca koju podižete sa računa mora biti manja od limita za podizanje.");

        stanje -= kol;
    }

    #endregion
}
```

Ostaje još da se definiše klasa *Banka*, koja predstavlja **kontejnersku klasu**. To zapravo znači da ova klasa sadrži liste sa elementima drugih klasa i omogućava izvođenje operacija nad tim listama (dodavanje, brisanje i izmjena). Definicija ove klase prikazana je u isječku koda ispod i vidljivo je da ova klasa ne posjeduje *properties* – razlog za to je što je manipulacija elementima liste dozvoljena isključivo kroz metode definisane u okviru ove klase. Također je prikazan i još jedan način za inicijalizaciju instanci klasa, gdje je pri otvaranju računa pozvan konstruktor kojim se postavlja početno stanje računa na 0, a zatim je pozvan *property* kojim se vrši izmjena ID-a računa.

```
public class Banka
{
    #region Atributi

    List<Klijent> klijenti = new List<Klijent>();
    List<Racun> racuni = new List<Racun>();

    #endregion

    #region Metode

    public void DodajKlijenta(Klijent osoba)
    {
        klijenti.Add(osoba);
    }

    public void OtvoriRacunZaOsobu(Klijent osoba)
    {
        Racun racun = new Racun(0)
        {
            Id = racuni.Count
        };
        racuni.Add(racun);
        osoba.Racuni.Add(racun);
    }

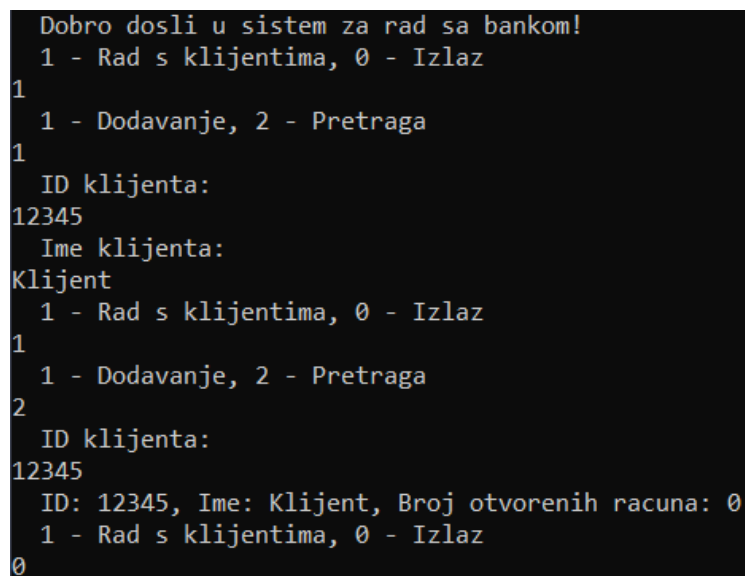
    public Klijent PronadiKlijenta(int identitet)
    {
        return klijenti.Find(osoba => osoba.Identitet == identitet);
    }

    public void PrebaciNovac(Racun racun, decimal kolicina)
    {
        racun.PoloziNovac(kolicina);
    }

    #endregion
}
```

Ovako definisan sistem klasa sada je moguće koristiti u okviru konzolne aplikacije koja je dodana kao drugi projekat u programsko rješenje. Način na koji se sve što je definisano može iskoristiti za rad sa korisnicima prikazan je u isječku koda ispod, pri čemu je korištenje aplikacije demonstrirano na Slici 4. Za kompleksniji rad studenti mogu dodati različite attribute i metode u klase kako bi se omogućilo povećavanje funkcionalnosti sistema klasa.

```
Banka banka = new Banka();
Console.WriteLine(" Dobro došli u sistem za rad sa bankom!");
string ulaz = "";
do
{
    Console.WriteLine(" 1 - Rad s klijentima, 0 - Izlaz");
    ulaz = Console.ReadLine();
    if (ulaz == "1")
    {
        Console.WriteLine(" 1 - Dodavanje, 2 - Pretraga");
        ulaz = Console.ReadLine();
        if (ulaz == "1")
        {
            Console.WriteLine(" ID klijenta:");
            int id = Int32.Parse(Console.ReadLine());
            Console.WriteLine(" Ime klijenta:");
            string ime = Console.ReadLine();
            Klijent klijent = new Klijent(id, ime);
            banka.DodajKlijenta(klijent);
        }
        else if (ulaz == "2")
        {
            Console.WriteLine(" ID klijenta:");
            int id = Int32.Parse(Console.ReadLine());
            Klijent pronađeni = banka.PronadiKlijenta(id);
            if (pronađeni == null)
                Console.WriteLine(" Traženi klijent ne postoji u bazi
podataka!");
            else
                Console.WriteLine(" ID: " + pronađeni.Identitet + ",
Ime: " + pronađeni.Ime + ", Broj otvorenih računa: " + pronađeni.Racuni.Count);
        }
    }
} while (ulaz != "0");
```



```
Dobro dosli u sistem za rad sa bankom!
1 - Rad s klijentima, 0 - Izlaz
1
1 - Dodavanje, 2 - Pretraga
1
ID klijenta:
12345
Ime klijenta:
Klijent
1 - Rad s klijentima, 0 - Izlaz
1
1 - Dodavanje, 2 - Pretraga
2
ID klijenta:
12345
ID: 12345, Ime: Klijent, Broj otvorenih racuna: 0
1 - Rad s klijentima, 0 - Izlaz
0
```

Slika 4. Način rada aplikacije za banku

3. Hijerarhija nasljeđivanja u C#

Veoma često dolazi do potrebe da se dio programskog koda koji je već napisan u nekoj od postojećih klasa ponovo iskoristi i nadogradi za druge primjene. Ukoliko je nadogradnja postojećeg koda takva da je potrebno kreirati nove klase koje u osnovi predstavljaju neke od postojećih klasa, onda je potrebno koristiti **hijerarhiju nasljeđivanja klasa**. Ovo zapravo znači da je potrebno napraviti novu klasu koja će naslijediti postojeću klasu i sve njene attribute i metode uz proizvoljne nadogradnje. Veoma je važno napomenuti da naslijeđena klasa uvijek treba biti zamjenjiva osnovnim tipom klase koju nasljeđuje.

Principi nasljeđivanja klasa biti će demonstrirani na primjeru koji je urađen u prethodnom dijelu laboratorijske vježbe. Neka je potrebno omogućiti korištenje privremenih transakcijskih računa koji važe 30 dana od dana otvaranja računa. Ovi transakcijski računi trebaju omogućiti i produženje roka trajanja, koje se odobrava samo ukoliko se produženje vrši u posljednjih 7 dana važenja računa, kao i ukoliko u tom trenutku na stanju ima više od 500 KM. Transakcije na ovim računima (podizanje i uplata novca) mogu se vršiti samo između 8 i 16 h.

Prvenstveno je potrebno izvršiti neophodne izmjene u klasi *Racun* koju nova vrsta računa **PrivremeniRacun** treba naslijediti. U tu svrhu potrebno je označiti sve privatne attribute kao **protected**, kako bi se mogli koristiti u okviru naslijeđene klase. Kako nova klasa posjeduje novu programsku logiku za podizanje i uplatu novca, ove metode u klasi *Racun* potrebno je proglasiti **virtualnim**, na način prikazan u isječku koda ispod. Na ovaj način omogućava se da se izvrši nova implementacija ovih metoda u izvedenoj klasi nove vrste računa.

```
#region Atributi

protected static decimal limitZaPodizanje = 1000;
protected decimal stanje;
protected int id;

#endregion

#region Metode

public virtual void PoloziNovac(decimal kol)
{
    if (kol < 0)
        throw new Exception("Količina novca koju polažete na račun mora biti pozitivna.");
    stanje += kol;
}

public virtual void DigniNovac(decimal kol)
{
    if (kol < 0)
        throw new Exception("Količina novca koju podižete sa računa mora biti pozitivna.");
    else if (kol > limitZaPodizanje)
        throw new Exception("Količina novca koju podižete sa računa mora biti manja od limita za podizanje.");
    stanje -= kol;
}

#endregion
```

Sada je potrebno definisati klasu *PrivremeniRacun*. Ova klasa treba naslijediti klasu *Racun*, što se definiše u samoj deklaraciji klase. Potrebno je definisati dva nova atributa za početak i kraj roka važenja računa, kao i novu metodu *ProdužiTrajanje* za produženje trajanja računa u slučaju ispunjenja neophodnih uslova. Osim toga, potrebno je i definisati nove implementacije za metode za polaganje i dizanje novca sa računa koje se označavaju ključnom riječi **override**. Konstruktor mora primati iste parametre kao i osnovni konstruktor (s tim da je dozvoljeno da prima i više parametara) jer se konstruktor osnovne klase nasljeđuje. Cijela definicija klase sa svim prethodno objašnjenim principima prikazana je u isječku koda ispod.

```
public class PrivremeniRacun : Racun
{
    #region Atributi

    DateTime pocetak, kraj;

    #endregion

    #region Konstruktor

    public PrivremeniRacun(decimal pocetnoStanje) : base(pocetnoStanje)
    {
        pocetak = DateTime.Now;
        kraj = DateTime.Now.AddMonths(1);
    }

    #endregion

    #region Metode

    public override void PoloziNovac(decimal kol)
    {
        if (kol < 0 || DateTime.Now.Hour < 8 || DateTime.Now.Hour > 15)
            throw new Exception("Količina novca koju polažete na račun mora biti pozitivna.");
        stanje += kol;
    }

    public override void DigniNovac(decimal kol)
    {
        if (kol < 0 || DateTime.Now.Hour < 8 || DateTime.Now.Hour > 15)
            throw new Exception("Količina novca koju podižete sa računa mora biti pozitivna.");
        else if (kol > limitZaPodizanje)
            throw new Exception("Količina novca koju podižete sa računa mora biti manja od limita za podizanje.");
        stanje -= kol;
    }

    public void ProduziTrajanje()
    {
        if (kraj.AddDays(-7) <= DateTime.Now && stanje >= 500)
            kraj = kraj.AddDays(30);
    }

    #endregion
}
```


Objektno Orijentisana Analiza i Dizajn

Veoma često dolazi do situacije kada je potrebno ponovo iskoristiti postojeći kod, ali to nije moguće izvršiti uz poštovanje principa nasljeđivanja. Ovo je posebno važno u slučajevima kada je potrebno omogućiti da se neka metoda poziva nad objektom bez obzira na njegov tip, kao i kada se u jednu kolekciju podataka želi smjestiti više različitih vrsta podataka. U tom slučaju neophodno je koristiti **interfejs** (*interface*) koji će omogućiti da više klasa koje nemaju logičku međusobnu povezanost imaju zajednički osnovni tip, kao i da koriste postojeće implementacije zajedničkih metoda.

U prethodnom primjeru, klase *Racun* i *Klijent* imaju atribut koji predstavlja jedinstveni identifikator. Bilo bi korisno posjedovati mogućnost za generisanje sekvence od 10 nasumičnih brojeva koja će se zatim dodijeliti ovim atributima u konstruktorima klasa. Također bi bilo korisno posjedovati mogućnost za generisanje opisa instanci ovih klasa. U tu svrhu potrebno je definisati interfejs **IPodaci** koji će posjedovati dvije metode – **GenerišiID** i **DajOpis**. Metoda za generisanje ID-a imati će *default* implementaciju koju će pozivati klasa *Banka* pri kreiranju novih računa i klijenata, a drugu metodu će ove klase zasebno implementirati. Definicija interfejsa prikazana je u isječku koda ispod.

```
public interface IPodaci
{
    int GenerišiID()
    {
        int broj = 0;
        for (int i = 0; i < 10; i++)
        {
            Random generator = new Random();
            broj += (int) Math.Pow(10, i) * generator.Next(0, 9);
        }
        return broj;
    }

    string DajOpis();
}
```

Sada je potrebno izmijeniti implementaciju postojećih klasa. U klasi *Racun* potrebno je dodati nasljeđivanje interfejsa (što se čini u samoj deklaraciji klase, na isti način kao i u prethodnom primjeru sa nasljeđivanjem klasa), a zatim je potrebno implementirati metodu *DajOpis*. Isto je izvršeno i prikazano u isječku koda ispod.

```
public class Racun : IPodaci
{
    ...

    public string DajOpis()
    {
        return "ID: " + id + ", Stanje: " + stanje;
    }

    ...
}
```

```
public class Klijent : IPodaci
{
    ...

    public string DajOpis()
    {
        return "ID: " + identitet + ", Ime: " + ime + ", Broj računa: " +
        racuni.Count;
    }

    ...
}
```

Posljednja izmjena koja se treba izvršiti je promjena klase *Banka*. Sada nije potrebno imati dvije, nego jednu kolekciju objekata kojima će se obuhvatiti i klijenti i računi preko baznog tipa – interfejsa *IPodaci*. Metoda za dodavanje novog klijenta sada se može generalizirati tako da se dozvoljava i kreiranje novog računa, jer se parametar koji je općeg tipa dodaje u kolekciju, koja je također općeg tipa. Pri dodavanju novog računa sada se može pozvati metoda *GenerišiID* (jer je račun definisan kao parametar općeg tipa).

Važno je napomenuti da je, ukoliko se žele koristiti metode i atributi koje posjeduje samo izvedena klasa, potrebno prethodno izvršiti konverziju općeg u izvedeni tip klase. Tako se parametar koji je tipa *IPodaci* pretvara u tip *Klijent* koristeći sintaksu **(Klijent)osoba**, a zatim je potrebno dodati još jedan set zagrada zbog prioriteta operacije `.` nad operacijom pretvaranja u objekat neke izvedene klase. **((Klijent)osoba)** omogućava da se parametar tipa *IPodaci* koristi kao izvedena klasa. U slučaju da konverziju ne bude moguće izvršiti, doći će do pojave izuzetka, pa se preporučuje korištenje *try-catch* blokova za provjeru ispravnosti poslanih podataka. Sve prethodno opisano prikazano je u isječku koda ispod.

```
public class Banka
{
    #region Atributi

    List<IPodaci> objekti = new List<IPodaci>();

    #endregion

    #region Metode

    public void DodajKlijentaIliRacun(IPodaci objekat)
    {
        objekti.Add(objekat);
    }

    public void OtvoriRacunZaOsobu(IPodaci osoba)
    {
        IPodaci racun = new Racun(0)
        {
            Id = osoba.GenerišiID()
        };
        objekti.Add(racun);
        ((Klijent)osoba).Racuni.Add((Racun)racun);
    }
}
```

```
public IPodaci PronadiKlijenta(int identitet)
{
    return objekti.Find(osoba => osoba is Klijent &&
((Klijent)osoba).Identitet == identitet);
}

public void PrebaciNovac(Racun racun, decimal kolicina)
{
    racun.PoloziNovac(kolicina);
}

#endregion
}
```

Na ovaj način omogućava se nasljeđivanje definicija i implementacija metoda od strane klase koje nemaju nikakve međusobne povezanosti. Ovime se lokaliziraju promjene u kodu i olakšava dodavanje novih klasa i funkcionalnosti u programsko rješenje. Principi korištenja apstraktnih klasa, nasljeđivanja na više nivoa i drugih mogućnosti programskog jezika C# ostavljaju se studentima za samostalno istraživanje.

4. Zadaci za samostalni rad

Programski kod aplikacije koja je kreirana u vježbi dostupan je u repozitoriju na sljedećem linku: <https://github.com/ehlymana/OOADVjezbe>, na *branchu* **lv4**.

1. Banka iz primjera u laboratorijskoj vježbi je proširila svoj biznis i sada omogućava otvaranje i štednih računa za korisnike (pored osnovnih i privremenih transakcijskih računa). Štedni račun sadrži dodatnu informaciju o valuti. Tu nije kraj promjenama - predstavnici banke očekuju da će se uskoro omogućiti i pravnim licima da koriste sve usluge banke kao i fizička lica. Pravno lice opisano je nazivom i adresom. Potrebno je prilagoditi sistem klasa tako da svi novi zahtjevi budu implementirani i sve nove funkcionalnosti omogućene.
2. Dodati u programsko rješenje za banku novu mogućnost: moguće je pronaći sve osobe koje su klijenti banke na osnovu dijela imena. Dodajte u programsko rješenje za banku i još jednu mogućnost: moguće je pronaći imena svih osoba koje su klijenti banke.
3. Razviti programsko rješenje koje će omogućiti za agenciju Stan vršenje evidencije stanova i mjesečni obračun izdavanja stana. Stanovi mogu biti namješteni i nenamješteni. Za sve stanove se navodi površina (broj kvadrata) stana, lokacija stana (gradsko ili prigradsko područje) i da li postoji internet konekcija. U slučaju da je stan namješten, navodi se ukupna vrijednost namještaja i broj kućanskih aparata u stanu. Obračun najma se vrši tako što zainteresovana osoba (klijent agencije Stan) navede rang vrijednosti za površinu stana koji želi iznajmiti i lokaciju stana. Nakon toga se prikažu svi stanovi koji odgovaraju navedenim osobinama, sa dodatnim podacima o tome da li je stan namješten i da li ima internet konekciju. Nakon toga klijent bira stan i vrši se obračun cijene izdavanja za prvi mjesec. Bez obzira da li je stan nenamješten ili namješten, ako pripada gradskom području osnovna cijena izdavanja je 200 KM mjesečno, a ako pripada prigradskom području, cijena izdavanja je 150 KM. Cijena po kvadratu stana za sve stanove je 1 KM. Na osnovnu cijenu izdavanja se dodaje broj kvadrata pomnožen sa jediničnom cijenom kvadrata stana. Za nenamješten stan se nakon toga cijena izdavanja povećava za 2 % ukoliko stan ima internet konekciju, a za namješten za 1 %. U slučaju da je stan namješten, izračunata cijena izdavanja se nadalje povećava tako što se doda 1 % od ukupne vrijednosti namještaja ukoliko je broj kućanskih aparata manji od 3, u suprotnom se dodaje 2% od ukupne vrijednosti namještaja. Obavijestiti korisnika o iznosu cijene izdavanja stana.
4. Proširiti rješenje za agenciju Stan tako da se može iznajmiti i luksuzni apartman koji na osnovnu cijenu od 1500 KM dodaje još cijenu zaposlenika koji rade na održavanju. Zaposlenik može biti batler, kuhar i vrtlar. Svaki zaposlenik ima svoje ime, prezime, datum zaposlenja i mjesečnu platu. Kuhar ima listu jela koja je u stanju da napravi. Vrtlar sadrži i stan u kom stanuje, s obzirom da agencija ima ugovor sa vrtlarima da uz posao dobiju i nenamješten stan. Batler ima pored osnovnih još i godine iskustva. Za potrebe pregleda imovine agencije, treba biti moguće ispisati sve stanove i zaposlenika agencije.