KONZORCIJ ŠOLSKIH CENTROV

REPUBLIKA SLOVENIJA
MINISTRSTVO ZA ŠOLSTVO IN ŠPORT

Naložba v vašo prihodnost
OPERACIJO DELNO FINANCIRA EVROPSKA UNIJA
Evropski socialni sklad

Načrtovanje programskih aplikacij

NPA

2.del

Srečo Uranič

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

KONZORCIJ ŠOLSKIH CENTROV
REPUBLIKA SLOVENIJA
MINISTRSTVO ZA ŠOLSTVO IN ŠPORT
Naložba v vašo prihodnost
OPERACIJO DELNO FINANCIRA EVROPSKA UNIJA
Evropski socialni sklad

SPLOŠNE INFORMACIJE O GRADIVU

Izobraževalni program

Tehnik računalništva

Ime modula

Načrtovanje programskih aplikacij – NPA 4

Naslov učnih tem ali kompetenc, ki jih obravnava učno gradivo

Vizuelno programiranje, dedovanje, knjižnice, večokenske aplikacije, delo z bazami podatkov, testiranje in dokumentiranje, nameščanje aplikacij.

Avtor: Srečo Uranič
Recenzent: Matija Lokar
Lektor: v lekturi
Datum: Julij 2011
CIP:

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

POVZETEK/PREDGOVOR

Gradivo ***Načrtovanje programskih aplikacij*** je namenjeno dijakom 4. letnika izobraževalnega programa SSI – Tehnik računalništva in dijakom 5 letnika PTI - Tehnik računalništva. Pokriva vsebinski del modula Načrtovanje in razvoj programskih aplikacij, kot je zapisan v katalogu znanja za ta program v poklicnem tehniškem izobraževanju. Kot izbrani programski jezik sem izbral programski jezik C# v razvojnem okolju Microsoft Visual C# 2010 Express. Gradivo ne vsebuje poglavij, ki so zajeta v predhodnem izobraževanju. Dostopna so v mojih mapah http://uranic.tsckr.si/ na šolskem strežniku TŠCKR (to so poglavja osnove za izdelavo konzolnih aplikacij, metode, varovalni bloki, razhroščevanje, tabele, zbirke, strukture, naštevanje, razredi in objekti, rekurzije, metoda main in datoteke). V gradivu bomo večkrat uporabljali kratico *OOP* – Objektno Orireniriranо Programiranje.

V besedilu je veliko primerov programov in zgledov, od najenostavnejših do bolj kompleksnih. Bralce, ki bi o posamezni tematiki radi izvedeli več, vabim, da si ogledajo tudi gradivo, ki je navedeno v literaturi.

Gradivo je nastalo na osnovi številnih zapiskov avtorja, črpal pa sem tudi iz že objavljenih gradiv, pri katerih sem bil "vpleten". V gradivu je koda, napisana v programskem jeziku, nekoliko osenčena, saj sem jo na ta način želel tudi vizualno ločiti od teksta gradiva.

Literatura poleg teoretičnih osnov vsebuje številne primere in vaje. Vse vaje, pa tudi številne dodatne vaje in primeri so v stisnjeni obliki na voljo na strežniku TŠC Kranj http://uranic.tsckr.si/VISUAL%20C%23/, zaradi prevelikega obsega pa jih v literaturo nisem vključil še več.

Programiranja se ne da naučiti s prepisovanjem tujih programov, zato pričakujem, da bodo dijaki poleg skrbnega študija zgledov in rešitev pisali programe tudi sami. Dijakom tudi svetujem, da si zastavijo in rešijo svoje naloge, ter posegajo po številnih, na spletu dosegljivih primerih in zbirkah nalog.

Gradivo ***Načrtovanje programskih aplikacij*** vsebuje teme: izdelava okenskih aplikacij, lastnosti in dogodki okenskih gradnikov, dogodki tipkovnice, miške in ure, kontrola in validacija uporabnikovih vnosov, sporočilna okna, dedovanje, virtualne in prekrivne metode, polimorfizem, knjižnice, pogovorna okna, delo s tiskalnikom, nadrejeni in podrejeni obrazci, dostop in ažuriranje podatkov v bazah podatkov, transakcije, testiranje in dokumentiranje aplikacije, izdelava namestitvenega programa.

Ključne besede: gradniki, lastnosti, metode, dogodki, validacija (preverjanje pravilnosti), sporočilna okna, pogovorna okna, razred, dedovanje, polimorfizem, virtual, override, using,

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

knjižnica, tiskalnik, baza, transakcija, MDI, DataSet, SQL, testiranje, dokumentiranje, namestitev.

Legenda: Zaradi boljše preglednosti je gradivo opremljeno z motivirajočo slikovno podporo, katere pomen je naslednji:

-  informacije o gradivu;
-  povzetek oz. predgovor;
-  kazala;
-  učni cilji;
-  napoved učne situacije;
-  začetek novega poglavja;
-  posebni poudarki;
-  nova vaja, oziroma nov primer;
-  rešitev učne situacije;
-  literatura in viri.

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

KAZALO

KAZALO SLIK:

KAZALO TABEL

UČNI CILJI

Učenje programiranja je privajanje na algoritmični način razmišljanja. Poznavanje osnov programiranja in znanje algoritmičnega razmišljanja je tudi nujna sestavina sodobne funkcionalne pismenosti, saj ga danes potrebujemo praktično na vsakem koraku. Uporabljamo oz. potrebujemo ga:

- pri vsakršnem delu z računalnikom;
- pri branju navodil, postopkov (pogosto so v obliki "kvazi" programov, diagramov poteka);
- za umno naročanje ali izbiranje programske opreme;
- za pisanje makro ukazov v uporabniških orodjih;
- da znamo pravilno predstaviti (opisati, zastaviti, ...) problem, ki ga potem programira nekdo drug;
- ko sledimo postopku za pridobitev denarja z bankomata;
- ko se odločamo za podaljšanje registracije osebnega avtomobila;
- za potrebe administracije – delo z več uporabniki;
- za ustvarjanje dinamičnih spletnih strani;
- za nameščanje, posodabljanje in vzdrževanje aplikacije;
- ob zbiranju, analizi in za dokumentiranje zahtev naročnika, komuniciranje in pomoč naročnikom;
- za popravljanje "tujih" programov

CILJI

- Spoznavanje oz. nadgradnja osnov programiranja s pomočjo programskega jezika C#.
- Poznavanje in uporaba razvojnega okolja Visual Studio za izdelavo programov.
- Načrtovanje in izdelava preprostih in kompleksnejših programov.
- Privajanje na algoritmični način razmišljanja.
- Poznavanje razlike med enostavnimi in kompleksnimi programskimi strukturami.
- Ugotavljanje in odpravljanje napak v procesu izdelave programov.
- Uporaba znanih rešitev na novih primerih.
- Spoznavanje osnov sodobnega objektno orientiranega programiranja.
- Manipuliranje s podatki v bazi podatkov.
- Testiranje programske aplikacije in beleženje rezultatov testiranja.
- Ob nameščanju, posodabljanju in vzdrževanju aplikacije.
- Ko zbiramo, analiziramo in dokumentiramo zahteve naročnika, komuniciramo z njim in

mu pomagamo.
- Izdelava dokumentacije in priprava navodil za uporabo aplikacije.
- Uporaba več modulov v programu in izdelava lastne knjižnice razredov in metod.
- Delo z dinamičnimi podatkovnimi strukturami.

SEZNAM KRVODAJALCEV

Za seznam krvodajalcev, urejen po krvnih skupinah, bi radi pripravili aplikacijo, ki bo omogočala vodenje njihovega seznama. Seznam naj bo možno dopolnjevati in urejati. Izdelali bomo večokensko aplikacijo, ter spoznali pojem dedovanja in polimorfizma. Naučili se bomo izdelati in uporabljati lastno knjižnico, spoznali pa še abstraktne razrede, abstraktne metode, *MDI* aplikacije, ter delo s tiskalnikom. V poglavju bo razložen tudi pojem *Garbage Colector*.

Dedovanje (inheritance)

Dedovanje (*Inheritance*) je eden od ključnih konceptov objektno orientiranega programiranja. Smisel in pomen dedovanja je v tem, da iz že zgrajenih razredov skušamo zgraditi bolj kompleksne. Dedovanje je mehanizem, s katerim se izognemo ponavljanju pri definiranju različnih razredov, ki pa imajo več ali manj značilnosti skupnih. Opredeljuje torej odnos med posameznimi razredi.

Vzemimo pojem *sesalec* iz biologije. Kot primer za sesalce vzemimo npr. konje in kite. Tako konji kot kiti počnejo vse kar počnejo sesalci nasploh (dihajo zrak, skotijo žive mladiče, …), imajo pa nekatere specifične značilnosti (konji imajo npr. štiri noge, ki jih kiti nimajo, imajo kopita …, obratno pa imajo npr. kiti plavuti, ki pa jih konji nimajo…). V Microsoft C# bi lahko za ta primer modelirali dva razreda: prvega bi poimenovali *Sesalec* drugega pa *Konj* in tretjega *Kit*. Ob tem bi deklarirali, da *Konj* deduje od *Sesalca*. Na ta način bi med sesalci in konjem vzpostavili povezavo v tem smislu, da so vsi konji sesalci. Obratno pa seveda ne velja! Podobno lahko deklariramo razred z imenom *Kit*, ki prav tako deduje iz razreda *Sesalec*. Objekti v razredu, ki deduje, imajo avtomatično vse lastnosti, ki jih imajo objekti v razredu, katerega se deduje, poleg njih pa še dodatne. Torej lastnosti, kot so npr, kopita ali pa plavuti pa lahko dodamo v razred *Konj* oz. razred *Kit*.

Osnovni razredi in izpeljani razredi

Deklaracija razreda, s katero želimo povedati, da razred deduje nek drug razred, ime naslednjo sintakso:

```
class IzpeljaniRazred : OsnovniRazred
```

```
{
    . . .
}
```

Izpeljani razred deduje od osnovnega razreda. Za razliko od nekaterih drugih programskih jezikov (npr. programskega jezika C++), lahko razred v programskem jeziku C# deduje največ od enega razreda. Seveda pa je lahko razred, ki podeduje nek osnovni razred, zopet podedovan v še bolj kompleksen razred.

Napišimo preprosti razred, s katerim predstavimo točko v dvodimenzionalnem koordinatnem sistemu. Denimo namreč, da ne želimo uporabiti vgrajenega razreda *Point*. Razred poimenujmo *Tocka*, predstavlja pa osnovni razred (razred, ki ga bomo dedovali), iz katerega bomo kasneje izpeljali razred *Krog*. Zaradi enostavnosti bomo vsa polja in metode označili kot *public* in zato ne bo potrebno pisati še lastnosti posameznih polj.

```
public class Tocka //Razred Tocka
{
    // koordinati točke
    public int x, y;
    //privzeti konstruktor
    public Tocka()
    {
        x = 0; y = 0;
    }
    // konstruktor
    public Tocka( int vrednostX, int vrednostY )
    {
        x = vrednostX;  y = vrednostY;
    }

    // metoda Opis vrne niz, ki predstavlja točko
    public string Opis()
    {
        return "[" + x + ", " + y + "]";
    }
} // konec razreda Tocka
```

Za razred *Krog* potrebujemo dva podatka: središče kroga in njegov polmer. Ker je središče kroga točka, je smiselno, da razred *Krog* deduje razred *Tocka*, dodamo pa mu še novo polje *polmer*.

```
// definicija razreda Krog
public class Krog : Tocka  //razred Krog deduje razred Tocka
{
    // dodatno polje: polmer kroga
    protected double polmer;
    public Krog()// privzeti konstruktor
    {
        /*tu se kliče PRIVZETI konstruktor osnovnega razreda Tocka, KI PA
```

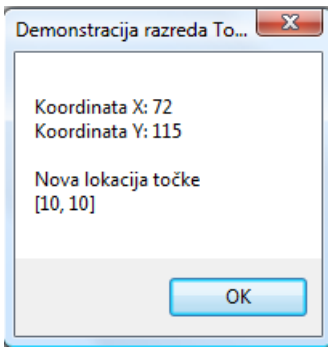
```
            MORA OBSTAJATI V RAZREDU TOCKA – to pa zaradi tega, ker v razredu
            Tocka obstaja tudi drug konstruktor s parametri !!!*/
            polmer = 0.0;
        }

        public Krog( int x, int y, double r ) // konstruktor
            : base( x, y )  //klic konstruktorja osnovnega razreda Tocka
        {
            polmer = r;
        }
        public double PloscinaKroga()// metoda vrne ploščino kroga
        {
            return Math.PI *polmer * polmer;
        }
        // metoda vrne niz, ki predstavi krog
        //še bolje public override Opis(razlaga v nadaljevanju)
        public string Opis()
        {
            return "Središče kroga = [" + x + ", " + y +"], polmer = " +
polmer+"\nPloščina: "+Math.Round(PloscinaKroga(),2);
        }
} // konec razreda Krog
```

Vemo že, da ima vsak razred vsaj en konstruktor in če ga ne napišemo sami, prevajalnik ustvari privzetega, brez parametrov. Izpeljani razred avtomatično deduje vsa polja osnovnega razreda, a ta polja je potrebno ob ustvarjanju novega objekta inicializirati. Zato mora konstruktor v izpeljanem razredu poklicati konstruktor svojega osnovnega razreda. V ta namen se uporablja rezervirana besedica *base*. Če dedujemo privzeti konstruktor, lahko besedico *base* izpustimo, saj je klic osnovnega privzetega konstruktorja avtomatski. Sicer pa jo uporabimo tako, da v glavi konstruktorja izpeljanega razreda dodamo *:base(parametri)*. Parametrov je toliko, kot je parametrov v osnovnem konstrukrorju. Tak klic smo uporabili tudi v zgornjem zgledu.

Če želimo v izpeljanem razredu napisati privzeti konstruktor, v osnovnem razredu pa je napisan vsaj en konstruktor s parametri, moramo privzetega napisati tudi v osnovnem razredu. Kadarkoli pa lahko v izpeljanem razredu kličemo poljuben konstruktor osnovnega razrada.

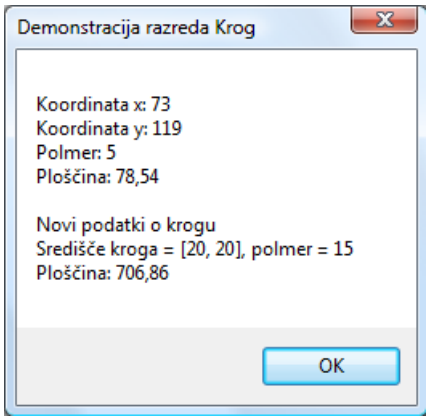
Ustvarimo sedaj po dva objekta vsakega razreda. Kodo zapišimo npr. v konstruktor nekega obrazca, ali pa v dogodek *Click* nekega gumba na poljubnem obrazcu.

```
Tocka t = new Tocka( 72, 115 );//objekt razreda Tocka
// trenutni vrednosti koordinat zapišemo v niz izpis
string izpis = "Koordinata X: " + t.x + "\nKoordinata Y: " + t.y;
t.x = 10; t.y = 10;   // Določimo novi koordinati

// nizu izpis dodamo še novi vrednosti koordinat
izpis += "\n\nNova lokacija točke\n" + t.Opis();
MessageBox.Show(izpis, "Demonstracija razreda Točka");
```



Slika 1: Predstavitev dveh objektov razreda *Tocka*.

```
Krog k = new Krog(73,119,5); //nov objekt razreda Krog
/*v niz izpis zapišimo trenutni vrednosti koordinat središča kroga, dodajmo
pa še polmer in ploščino*/
izpis = "Koordinata x: " + k.x + "\nKoordinata y: " + k.y + "\nPolmer: " +
k.polmer + "\nPloščina: " + Math.Round(k.PloscinaKroga(), 2);

// Določimo novo središče kroga in nov polmer
k.x = 20;
k.y = 20;
k.polmer=15;
// nizu izpis dodamo še nove vrednosti središča kroga in polmera
izpis += "\n\nNovi podatki o krogu\n" +  k.Opis();
MessageBox.Show(izpis,"Demonstracija razreda Krog");
```



Slika 2: Predstavitev dveh objektov razreda *Krog*.

Iz razreda *Krog* pa seveda zopet lahko izpeljimo poljuben dodatni razred, npr. *Valj*. Razred *Valj* deduje vse lastnosti in metode razreda *Krog* in seveda posredno s tem tudi vse lastnosti in metode razreda *Tocka*.

```
public class Valj : Krog
```

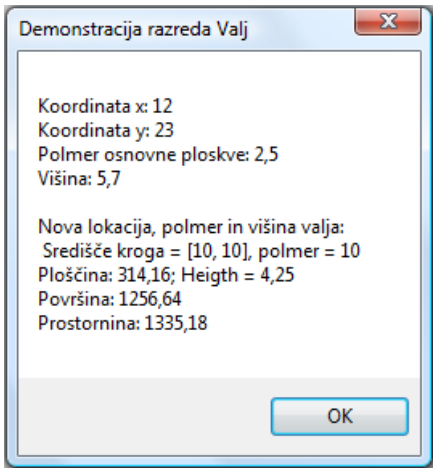
```
{
    public double visina; // dodatno polje razreda Valj

    public Valj()    // privzeti konstruktor
    {
        visina = 0.0;
    }
    // dodatni konstruktor
    public Valj( int x, int y,double polmer, double h )
        : base(x, y, polmer)// klic konstruktorja razreda Krog
    {
        visina = h;
    }

    // metoda, ki vrne površino valja
    public double Povrsina()
    {
        return 2 * base.PloscinaKroga() + 2 * Math.PI * polmer * polmer;
    }
    // metoda vrne prostornino valja
    public double Prostornina()
    {
        return base.PloscinaKroga() * visina;
        /*ali pa kar: return PloscinaKroga()*visina, saj metoda s tem
          imenom obstaja le v dedovanem rezredu Krog */
    }

    // metoda vrne niz, ki predstavi valj
    public string Opis()
    {
        return base.Opis() + "; Heigth = " + visina;
    }
} // konec razreda Valj
```

V zgornjem primeru smo v objektni metodi *Povrsina* poklicali metodo *PloscinaKroga* dedovanega razreda *Krog*. Uporabili smo besedico *base* in napisali *base.PloscinaKroga()*. Kadarkoli se namreč želimo v izpeljanem razredu sklicevati na neko metodo dedovanega razreda, pred imenom metode zapišemo besedico *base*. Z njo povemo, da se metoda (ali pa polje oz. lastnost) ki ji sledi, nahaja v razredu, ki je dedovan. Izpustimo jo lahko v primeru, da metoda z enakim imenom v izpeljanem razredu ne obstaja.

Ustvarimo še dva objekta tipa *Valj* in demonstrirajmo uporabo lastnosti in polj.

```
Valj valj = new Valj( 12, 23, 2.5, 5.7 );
// trenutni vrednosti objekta valj zapišemo v niz izpis
string output = "Koordinata x: " + valj.x +"\nKoordinata y: " + valj.y +
    "\nPolmer osnovne ploskve: "+ valj.polmer + "\nVišina: " + valj.visina;
// določimo nove koordinate, polmer in višino valja
valj.x = 10;
```

```
valj.y = 10;
valj.polmer = 10;
valj.visina = 4.25;

// nizu izpis dodamo še nove vrednosti
output +=
    "\n\nNova lokacija, polmer in višina valja:\n " +
    valj.Opis() + "\nPovršina: " +Math.Round(valj.Povrsina(),2) +
"\nProstornina: " + Math.Round(valj.Prostornina(),2);
    MessageBox.Show( output, "Demonstracija razreda Valj" );
```



Slika 3: Predstavitev dveh objektov razreda *Valj*.

Nove metode – operator new

Razredi lahko vsebujejo več ali manj metod in slej ko prej se lahko zgodi, da se pri dedovanju v izpeljanih razredih ime metode ponovi – v izpeljanem razredu torej napišemo metodo, katere ime, število in tipi parametrov se ujemajo z metodo bazičnega razreda. Tudi v zgornjih dveh primerih je bilo tako z metodo *Opis()*. Pri prevajanju bomo zato o tem dobili ustrezno opozorilo - *warning*. Metoda v izpeljanem razredu namreč v tem primeru prekrije metodo bazičnega razreda.

Program se bo sicer prevedel in tudi zagnal, a opozorilo moramo vzeti resno. Če namreč napišemo nek nov razred, ki bo podedoval razred *Krog*, bo uporabnik morda pričakoval, da se bo pri klicu metode *Opis* pognala metoda bazičnega razreda, a v našem primeru se bo zagnala metoda razreda *Valj*. Problem seveda lahko rešimo tako, da metodo *Opis* v izpeljanem razredu preimenujemo (npr *Izpis*), še boljša rešitev pa je ta, da v izpeljanem razredu eksplicitno povemo, da gre za NOVO metodo – to storimo tako , da v glavi metode pred označevalcem *public* zapišemo operator *new*.

```
new public string ToString()// glava metode
{
```

```
    // telo metode
}
```

Virtualne in prekrivne metode

Pogosto želimo metodo, ki smo je napisali v osnovnem razredu, v izpeljanih razredih skriti in napisati novo metodo, ki pa bo imela enako ime in enake parametre. Metodo, za katero želimo že v osnovnem razredu označiti, da jo bomo lahko v nadrejenih razredih nadomestili z novo metodo (jo prekrili z drugo metodo z enakim imenom), označimo kot virtualno (*virtual*), npr.:

```
/*virtualna metoda v bazičnem razredu – v izpeljanih razredih bo lahko
  prekrita(override)*/
public virtual string Opis()
{
    // telo metode
}
```

V nadrejenem razredu moramo v takem primeru pri metodi z enakim imenom uporabiti rezervirano besedico *override*, s katero povemo, da bo ta metoda prekrila/prepisala bazično metodo z enakim imenom in enakimi parametri.

```
/* izpeljani razred – besedica override pomeni, da smo s to metodo prekrili
   bazično metodo z enakim imenom*/
public override string Opis()
{
    // telo metode
}
```

Bededico *override* smo torej uporabili zaradi *dedovanja*. Z dedovanjem smo namreč avtomatično pridobili metodo *Opis*. To je razlog, da je ta metoda vedno na voljo v vsakem razredu, tudi če je ne napišemo. Če želimo napisati svojo metodo, ki se imenuje enako kot podedovana metoda, moramo pri deklaraciji uporabitii besedico *override*. S tem "povozimo" obstoječo metodo.

Prekrivanje metode (*overriding*) je torej mehanizem, kako izvesti novo implementacijo iste metode – *virtualne* in *override* metode so si v tem primeru sorodne, saj se pričakuje, da bodo opravljale enako nalogo, a nad različnimi objekti (izpeljanimi iz osnovnih razredov, ali pa iz podedovanih razredov).

Pri deklaraciji takih metod (pravimo jim tudi polimorfne metode) z uporabo rezerviranih besed *virtual* in *override*, pa se moramo držati nekaterih pomembnih pravil:

- Metoda tipa *virtual* oz. *override* NE more biti zasebna (ne more biti *private*), saj so zasebne metode dostopne le znotraj istega razreda.
- Obe deklaraciji metod, tako *virtualna* kot *override* morata biti identični: imeti morata enako ime, enako število in tip parametrov in enak tip vrednosti, ki jo vračata.

- Obe metodi morata imeti enak dostop. Če je npr. virtualna metoda označena kot javna (*public*), mora biti javna tudi metoda *override*.
- Prepišemo (prekrijemo/povozimo) lahko le virtualno metodo. Če metoda ni označena kot virtualna in bomo v nadrejenem razredu skušali narediti *override*, bomo dobili obvestilo o napaki.
- Če v nadrejenem razredu ne bomo uporabili besedice *override*, bazična metoda ne bo prekrita. To pa hkrati pomeni, da se bo tudi v izpeljanem razredu izvajala metoda bazičnega razreda in ne tista, ki smo napisali v izpeljanem razredu.
- Če neko metodo označimo kot *override*, jo lahko v izpeljanih razredih ponovno prekrijemo z novo metodo.

Drug način, kako neko metodo v izpeljanih razredih skriti, pa je uporaba operatorja *new*. Razlika med obema načinoma je v tem, da v primeru operatorja *new* osnovno metodo le skrijemo, v primeru prekrivanja pa jo seveda prekrijemo. Pojasnimo to na naslednjih dveh primerih. V prvem primeru bomo metodo osnovnega razreda s pomočjo operatorja *new* skrili, v drugem primeru pa jo bomo prekrili.

```
class OsnovniRazred
{
    //objektna metoda Izpis
     public void Izpis() { MessageBox.Show("OsnovniRazred->Izpis()"); }
}

class IzpeljaniRazred : OsnovniRazred
{
     //z uporabo operatoraja new originalno metodo Izpis SKRIJEMO
     public new void Izpis() { MessageBox.Show("IzpeljaniRazred->Izpis()"); }
}
```

Iz obeh razredov tvorimo tri objekte, in pokličimo metodo *Izpis*

```
// napoved novih objektov
OsnovniRazred a,c;
IzpeljaniRazred b;

a = new OsnovniRazred();
b = new IzpeljaniRazred();
a.Izpis();  // izpis --> "OsnovniRazred->Izpis()"
b.Izpis();  // izpis --> "IzpeljaniRazred->Izpis()"

//objekt c, ki je napovedan kot OsnovniRazred, izpeljimo iz IzpeljaniRazred
c = new IzpeljaniRazred();
c.Izpis();  // izpis --> "IzpeljaniRazred->Izpis()"
```

Pri objektih *a* in *b* je izpis je pričakovan, saj se izvede metoda razreda, iz katerega je objekt ustvarjen. Ker smo v izpeljanem razredu originalno metodo *Izpis* le skrili, se pri klicu metode metode *Izpis* objekta *c*, izvede metoda osnovnega razreda.

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

Še primer prekrivanja:

```
class Osnovni
{
    public virtual void Izpis() { MessageBox.Show("Osnovni->Izpis()"); }
}

class Izpeljani : Osnovni
{
     //z besedico override originalno metodo Izpis PREKRIJEMO
     public override void Izpis() { MessageBox.Show("Izpeljani->Izpis()"); }
}
```

Iz obeh razredov tvorimo objekta, in pokličimo metodi Izpis

```
// napoved novih objektov
Osnovni a;
Izpeljani b;

//ustvarjanje novih objektov
a = new Osnovni();
b = new Izpeljani();
a.Izpis();  // izpis --> "Osnovni->Izpis()"
b.Izpis();  // izpis --> "Izpeljani->Izpis()"
```

Obakrat se seveda izvede metoda razreda, iz katerega je objekt ustvarjen.

Če pa pri inicializaciji objekta, ki je tipa *Osnovni* uporabimo konstruktor razreda *Izpeljani*, se pri klicu metode *Izpis* izvede metoda izpeljanega razreda.

```
Osnovni c;
c = new Izpeljani();
c.Izpis();  // izpis --> "Izpeljani->Izpis()"
```

To pa je osnova načela, ki se imenuje p*olimorfizem*. Izvede se metoda razreda, iz katerega je objekt izpeljan.

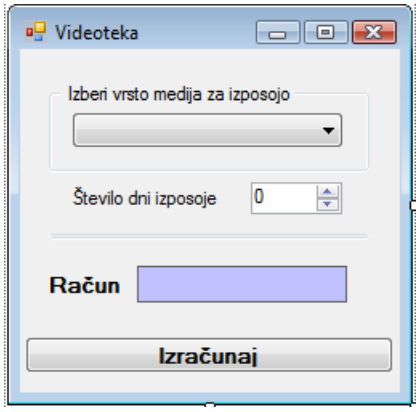
Videoteka

Kolega je odprl videoteko za izposojo CD-jev, video posnetkov in DVD–jev. Za izposojo posameznega medija si je zamislil poseben algoritem za znesek izposoje:

- pri izposoji CD-jev je znesek za prve tri dni različen, za vsak naslednji dan pa se poveča za enak znesek;
- pri izposoji video posnetkov se dnevna cena v primerjavi z izposojo CD-ja poveča za nek dodatek;

- pri izposoji DVD-jev se dnevna cena v primerjavi z izposojo CD-ja poveča za nek drug dodatek, končnemu znesku pa se doda še posebna kavcija (10 €).

Izdelati želimo preprosto aplikacijo, ki bo kolegu omogočila izračun zneska za izposojene medije. V ta namen najprej pripravimo obrazec in na njem *ComboBox* za izbiro medija, in gradnik *NumericUpDown* za vnos števila dni izposoje. Temu gradniku pustimo privzeto lastnost *minimum* enako 0, lastnost *maximum* pa nastavimo na poljubno, npr. 365 (maksimalno število dni izposoje bo eno leto). Dodajmo še oznako za izpis rezultata in gumb za izračun rezultata.



Slika 4: Gradniki na obrazcu Videoteka.

Za izračun zneska izposoje posameznih medijev napišimo razred *RacunCD*, ki ga bomo kasneje dedovali v razredu *RacunVideo*, tega pa v razredu *RacunDVD*. V naši rešitvi bodo zneski izposoje, dodatki in kavcija določeni vnaprej, lahko pa bi jih imeli shranjene v neki datoteki. V izpeljanih razredih bomo za izračun izposoje uporabili kar metodo osnovnega razreda, znesek pa bomo nato ustrezno povečali. Koda bo zaradi tega krajša in bolj pregledna.

V osnovnem razredu bomo konstantna polja označili kot *protected*. Če je neko polje, ali pa metoda označena kot *protected*, je zasebna v tem razredu, a javna v razredu, ki le-tega deduje.

```
public class RacunCD
{
    /*konstante so protected, kar pomeni da so v tem razredu zasebna,
      v dedovanih razredih pa javna*/
    protected const double enDan = 1.00;//znesek za 1 dan izposoje
    protected const double dvaDni = 2.50;//znesek za 2 dni izposoje
    protected const double triDni = 4.00;//znesek za 3 dni izposoje
    //znesek za vsak dan izposoje, če je število dni več kot 3
    protected const double vecDni = 1.50;

    //virtualna metoda za izračun zneska izposoje
    public virtual double Skupaj(int dni)
    {
        double racun=0;
        if (dni > 3)
        {
```

```
            racun = (dni - 3) * vecDni + triDni;
        }
        else
        {
            switch (dni)
            {
                case 1:
                    racun = enDan; break;
                case 2:
                    racun = dvaDni; break;
                case 3:
                    racun = triDni; break;
            }
        }
        return racun;
    }
}
```

Razred *RacunVideo* ima še dodatno polje *dodatek*, to je dodatni znesek za vsak dan izposoje. Pri izračunu zneska izposoje uporabimo metodo osnovnega razreda, dodamo pa še znesek dodatka za vsak dan posebej.

```
//razred RacunVideo deduje razred RacunCD
public class RacunVideo: RacunCD
{
    //razred RacunVideo deduje vsa konstantna polja razreda RacunCD
    //dodano je novo polje dodatek
    protected double dodatek;
    public RacunVideo(double dodatek)
    {
        this.dodatek = dodatek;
    }

    //metoda za izračun zneska izposoje prekrije metodo osnovnega razreda
    public override double Skupaj(int dni)
    {
        /*uporabimo kar metodo Skupaj osnovnega razreda, dodamo pa še znesek
          Dodatka za vsak dan izposoje*/
        return base.Skupaj(dni) + dni * dodatek;
    }
}
```

Razredu *RacunDVD* dodamo še polje *kavcija*, to je dodatni enkratni znesek. Pri izračunu zneska izposoje zopet uporabimo metodo dedovanega razreda, dodamo pa še kavcijo.

```
//razred RacunDVD deduje razred RacunVideo
public class RacunDVD : RacunVideo
{
    //razred RacunDVD deduje vsa konstantna polja razreda RacunVideo
```

```
    //dodano je novo polje - enkratni znesek kavcije
    private double kavcija;
    public RacunDVD(double dodatek, double kavcija): base(dodatek)
    {
        this.kavcija = kavcija;
    }

    //metoda za izračun zneska izposoje prekrije metodo osnovnega razreda
    public override double Skupaj(int dni)
    {
        //pokličemo metodo Skupaj dedovanega razreda, dodamo pa še kavcijo
        return base.Skupaj(dni) +kavcija;
    }
}
```

Napisane razrede moramo še preizkusiti. V konstruktorju že pripravljenega obrazca dodajmo tri postavke v gradnik *ComboBox*. Program opremimo še z lastno logično metodo Vnos_Validating. Ta metoda vrne False v primeru, da uporabnik pred izračunom ne izbere medija in števila dni za izposojo.

```
public Form1()
{
    InitializeComponent();
    //postavke dodamo v ComboBox
    comboBox1.Items.Add("CD");
    comboBox1.Items.Add("Video");
    comboBox1.Items.Add("DVD");
}

private bool Vnos_Validating()
{
    if (comboBox1.Text == "")
    {
        MessageBox.Show("Izberi vrsto medija za izposojo!", "MEDIJ?");
        return false;
    }
    else if (numericUpDown1.Value == 0)
    {
        MessageBox.Show("Število dni izposoje mora biti večje od 0!",
                        "ŠTEVILO DNI!");
        return false;
    }
    else return true;
}
```

Končno napišimo še odzivno metoda dogodka *Click* za gumb *Izračun*. Metoda bo izračunani znesek izposoje zapisala v oznako ob napisu *Račun*.

```
private void button1_Click(object sender, EventArgs e)
{
    if (Vnos_Validating())
    {
        label3.Text = "";
        if (comboBox1.SelectedIndex == 0)
        {
            RacunCD racun = new RacunCD();
            label3.Text = racun.Skupaj((int)numericUpDown1.Value).ToString();
        }
        else if (comboBox1.SelectedIndex == 1)
        {
            //račun za video se vsak dan poveča za 0.5 EUR
            RacunVideo video = new RacunVideo(0.5);
            label3.Text = video.Skupaj((int)numericUpDown1.Value).ToString();
        }
        else if (comboBox1.SelectedIndex == 2)
        {
            /*račun za DVD se vsak dan poveča za 1.1 EUR, dodana pa je še
              posebna kavcija*/
            RacunDVD dvd = new RacunDVD(1.1, 10);
            label3.Text = dvd.Skupaj((int)numericUpDown1.Value).ToString();
        }
        label3.Text = "€ " + label3.Text;
    }
}
```

Vsi objekti izpeljani iz razredov *RacunCD*, *RacunVideo* in *RacunDVD* poznajo objektno metodo *Skupaj*, a se nanjo odzivajo različno (izračun je drugačen glede na to, ali gre za CD, video ali DVD). Zmožnost, da se metoda z enakim imenom različno odziva glede na to, iz katerega razreda je izpeljana, je prav tako eden izmed ključnih konceptov objektnega programiranja, ki se imenuje *polimorfizem*. V splošnem pojem *polimorfizem* označuje dejstvo, da se iz tipa objekta ugotovi, katero različico metode je potrebno uporabiti.

Dedovanje vizuelnih gradnikov

Z dedovanjem vizuelnih gradnikov smo se srečevali že doslej, a se tega nismo zavedali. Pri vsakem novem projektu je razvojno okolje ustvarilo nov obrazec tako, da je ta podedoval razred imenovan *Form*. *Form* je vnaprej pripravljen splošni obrazec, to je okno, ki vsebuje naslovno vrstico s sistemskimi gumbi, ter delovno površino. Projekt smo gradili tako, da smo na delovno površino obrazca postavljali gradnike. Ti gradniki so dejansko postali lastnosti (komponente, oziroma polja) tega razreda. Za vsak na novo postavljen gradnik na obrazcu, je razvojno okolje avtomatično dodalo ustrezno kodo, ki ustvari objekt (vizualni gradnik) v metodo *InitializeComponent()*. Ta se nahaja v drugem delu razreda *Form1*, v datoteki *Form1.Designer.cs*. Če smo npr. na obrazec postavili gradnika tipa *Label* in *TextBox*, nam je razvojno okolje v tej datoteki ustvarilo kodo

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox textBox1;
```

Poleg teh dveh stavkov, se v metodi *InitializeComponent* pojavijo še "avtogenerirani" stavki, ki ustvarijo dva nova objekta in v katerih so določene osnovne oblikovne značilnosti teh dveh objektov na obrazcu.

```
this.label1 = new System.Windows.Forms.Label();//nov objekt tipa Label
this.textBox1 = new System.Windows.Forms.TextBox();//nov objekt tipa TextBox
//
// label1
//
this.label1.AutoSize = true;
this.label1.Location = new System.Drawing.Point(22, 23); //položaj labele
this.label1.Name = "label1"; //ime labele
this.label1.Size = new System.Drawing.Size(35, 13); //velikost labele
this.label1.TabIndex = 0; //zaporedna številka gradnika na obrazcu
this.label1.Text = "label1";//napis na labeli
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(25, 55); //položaj
this.textBox1.Name = "textBox1"; //ime TextBox-a
this.textBox1.Size = new System.Drawing.Size(100, 20); //velikost
this.textBox1.TabIndex = 1;
```

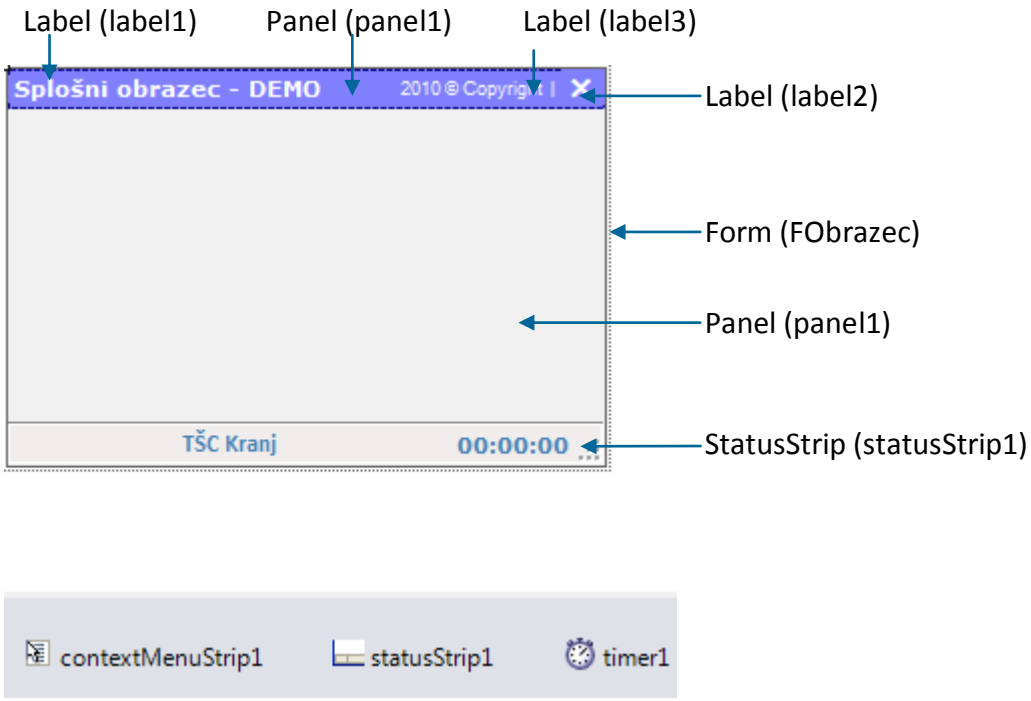
V telo razreda *Form1* datoteke *Form1.cs*, pa smo že doslej pisali odzivne dogodke in metode.

```
public partial class Form1 : Form  //Form1 je izpeljan iz razreda Form
{
    public Form1()//Konstruktor razreda Form1
    {
       InitializeComponent();
    }
    //Odzivni  dogodki  in metode razreda Form1
}
```

Pri izdelavi projektov pa slej ko prej pride tudi do situacije, da moramo izdelati obrazec, ki je zelo podoben (ali pa celo enak) obrazcu, ki smo ga naredili v enem od prejšnjih projektov. Obrazec, ki ga bomo potrebovali v več projektih, lahko zato pripravimo vnaprej. Na njem označimo gradnike, ki jih v izpeljanih obrazcih ne bo možno spreminjati (v oknu *Properties* nastavimo *Modifiers* na *private*) in tiste, ki jih bomo lahko spreminjali (*modifiers* nastavimo kot *protected*). Čeprav se oznaka *Modifiers* nahaja v oknu *Properties* pod zavihkom lastnosti, pa dejansko označuje način dostopnosti do tega objekta. Vsaka sprememba te "lastnosti" povzroči tudi spremembo "avtogenerirane" kode v datoteki *Designer.cs*.

Ko je obrazec in gradniki na njem pripravljen, ga še prevedemo, nato pa ga v novih projektih enostavno dedujemo in s tem nadomestimo splošni obrazec imenovan *Form*.

Za vajo izdelamo svoj bazični obrazec, ki ga bomo nato podedovali v novem projektu. Odprimo nov projekt in ga poimenujmo *Obrazec*. Obrazec *Form1* preimenujmo v *FObrazec*, nanj pa postavimo nekaj gradnikov, tako kot kaže spodnja slika. Gradnike postavljajmo na obrazec v enakem zaporedju, kot kaže spodnja tabela.



Slika 5: Gradniki bazičnega obrazca *FObrazec*.

Gradnik	Lastnost	Nastavitev	Opis
FObrazec	ControlBox	False	Na obrazcu ni sistemskih gumbov
	FormBorderStyle	None	Obrazec nima okvirja
	Size	300 x 200	Velikost obrazca
	StartPosition	CenterScreen	Obrazec se bo odprl na sredini zaslona
	TopMost	True	Obrazec se bo vedno odprl nad vsemi ostalimi obrazci, ki nimajo te lastnosti nastavljene na

			TopMost=True
statusStrip1	BackColor	Transparent	Barva ozadja
	Font	Verdana;8,25pt	
	Items: • toolStripLabel1 ○ Font	Calibri;9pt;style=Bold	Pisava oznake
	Items: • toolStripLabel1 ○ ForeColor	SteelBlue	Barva oznake
	Items: • toolStripLabel1 ○ Spring	True	Oznaka zasede vso prosto širino gradnika statusStrip1
	Items: • toolStripLabel1 ○ Text	TŠC Kranj	Vsebina oznake
	Items: • toolStripLabel2 ○ Font	Calibri;9pt;style=Bold	Pisava oznake
	Items: • toolStripLabel2 ○ ForeColor	SteelBlue	Barav oznake
	Items: • toolStripLabel1 ○ Text	00:00:00	Začetna oznaka
panel2	BackColor	128; 128; 255	Barva ozadja
	Dock	Top	Plošča bo pripeta pod vrhom obrazca
label1	Anchor	Top, Left	Oznaka bo vedno sidrana na levem robu obrazca
	BackColor	128; 128; 255	Barva ozadja
	Font	Verdana;8,25pt; style=Bold	Pisava
	ForeColor	White	Barva znakov

	Modifiers	Protected	Oznako bomo na podedovanem obrazcu lahko spremenili
	Text	Splošni obrazec-DEMO	Napis na oznaki
label2	Anchor	Top, Right	Oznaka bo vedno sidrana na desnem robu obrazca
	BackColor	128; 128; 255	Barva ozadja
	Font	Verdana;12pt; style=Bold	Pisava
	ForeColor	White	Barva znakov
	Text	x	Znak 'x' (za zapiranje okna)
label3	Anchor	Top, Right	Oznaka bo vedno sidrana na desnem robu obrazca
	BackColor	Transparent	Barva ozadja
	Font	MicrosoftSans Serif; 6,75pt	Pisava
	ForeColor	White	Barva znakov
	Text	2010 © Copyright |	Napis na oznaki
panel1	BackColor	Transparent	Barva ozadja
	BorderStyle	FixedSingle	Okvir plošče
	Dock	Fill	Plošča je raztegnjena čez celoten obrazec
	ContextMenuStrip	ContextMenuStrip	Lebdeči meni
	Modifiers	Protected	Na izpeljanih obrazcih bomo lahko dodajali nove gradnike
timer1	Enabled	True	Timer je omogočen
	Interval	1000	Timer se sproži vsako sekundo
contextMen	Items	Info	Napis na oznaki menija

uStrip1	▸ toolStripMenuItem1 ○ Text		
	Modifiers	Protected	Lebdeči meni bomo lahko na podedovanih obrazcih dopolnjevali

Tabela 1: Lastnosti gradnikov bazičnega obrazca.

Napisati moramo še nekaj odzivnih metod: to so dogodki oznake *label2*, ki ji bomo dodelili vlogo zapiranja obrazca.

```
public partial class FObrazec : Form
{
    public FObrazec()
    {
        /*POZOR! V izpeljanem obrazcu NE MOREMO spreminjati (četudi so
označeni kot Protected!) gradnikov MenuStrip in StatusStrip lahko pa
spreminjamo npr. ContextMenuStrip!!!*/
        InitializeComponent();
    }
    private void label2_Click(object sender, EventArgs e)
    {
        /*zapiranje bazičnega obrazca (in s tem tudi vseh izpeljanih
          obrazcev)*/
        Close();
    }
    private void label2_MouseDown(object sender, MouseEventArgs e)
    {
        label2.ForeColor = Color.Red;//ob kliku miške se barva spremeni
    }
    private void label2_MouseEnter(object sender, EventArgs e)
    {
        //barva ob vstopu miške v območje oznake
        label2.ForeColor = Color.DarkRed;
    }
    private void label2_MouseLeave(object sender, EventArgs e)
    {
        //ko z miško zapustimo oznako, je barva enaka kot pred vstopom
        label2.ForeColor = Color.White;
    }
    private void timer1_Tick(object sender, EventArgs e)
    {
        //prikaz časa na oznaki statusne vrstice
        toolStripStatusLabel2.Text = DateTime.Now.ToLongTimeString();
    }
}
```

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

Tako pripravljen obrazec shranimo in prevedemo. Bazični obrazec je sedaj pripravljen za dedovanje. Odprimo nov projekt in ga poimenujmo *DedovanjeObrazca*. Projektu moramo najprej omogočiti dostop do bazičnega obrazca: v oknu *SolutionExplorer* desno kliknimo na *References* in izberimo *Add Reference*. Odpre se okno *Add Reference*, v katerem izberemo zavihek *Browse* in v njem poiščemo datoteko *Obrazec.exe*, ki je nastala kot rezultat izdelave bazičnega obrazca (nahaja se seveda v mapi *Bin→Debug* projekta *Obrazec*). Izbiro potrdimo s klikom na gumb *OK*. V oknu *Solution Explorer* se v mapi *References* pojavi nov element z imenom *Obrazec*, do katerega imamo sedaj poln dostop.

Preklopimo na *CodeView* našega projekta in ime splošnega podedovanega obrazca (Form) nadomestimo z imenom dedovanega obrazca F*Obrazec*.

```
//dedovanje obrazca FObrazec
public partial class Form1 : FObrazec
{
    public Form1()
    {
       …
```

Če sedaj preklopimo na pogled *Design*, bomo namesto klasičnega začetnega obrazca že zagledali podedovani obrazec *FObrazec*, na katerega lahko polagamo nove gradnike in pišemo nove odzivne metode. Ker smo oznako *label1* na bazičnem obrazcu označili kot *Protected*, jo sedaj lahko poljubno spremenimo, na moremo pa spremeniti oznak *label2* in *label3*, ki sta na bazičnem obrazcu označeni kot *Private*. Prav tako ne moremo spremeniti statusne vrstice izpeljanega obrazca.

Višje verzije razvojnega okolja vključujejo tudi možnosti dedovanja s pomočjo posebne predloge imenovane *Inherit Form*. Če želimo v projektu dedovati nek že pripravljen obrazec, potem v glavnem meniju izberemo *Project→Add Windows Form…* in v prikazanem oknu izberemo *Inherited Form*. Izbiro potrdimo s klikom na gumb *Add*. Odpre se okno *Inheritane Picker*, kjer izberemo komponento ki jo dedujemo: to je izvršilna datoteka (*.exe*) že prej ustvarjenega obrazca znotraj naše rešitve (ali pa s klikom na gumb *Browse* poiščemo neko datoteko tipa dll). Izbiro potrdimo s klikom na gumb *OK*.

Kaj pa, če bomo obrazec, ki ga dedujemo v novih projektih, kadarkoli kasneje dopolnjevali, oz. spreminjali? Po spremembah ga moramo ponovno prevesti, prav tako pa je potrebno ponovno prevajanje vseh projektov, ki ta obrazec dedujejo. V primeru, da projekt razvijamo na drugem računalniku, moramo seveda nanj prenesti spremenjeni *dll* ali *exe*.

Izdelava lastne knjižnice razredov

V dosedanjih zgledih smo razrede pisali le za "lokalno uporabo", znotraj določenega programa. Razred (ali pa več razredov) pa lahko zapišemo tudi v svojo datoteko, ki jo potem dodajamo k različnim projektom, ali pa celo zgradimo svojo *knjižnico razredov*, ki jo bomo uporabljali v

različnih programih. Na ta način bomo tudi bolj ločili tisti del programiranja, ko gradimo razrede in tisti del, ko uporabljamo objekte določenega razreda.

Naučimo se najprej, kako zgradimo svojo knjižnico razredov, v kateri bomo napisali nekaj razredov. Za ustvarjanje novega projekta tokrat ne *izberemo Windows Forms Applications*, ampak *Class Library*. Kot ime knjižnice zapišimo *MojaKnjiznica*. *Visual C#* je za nas pripravil imenski prostor *MojaKnjiznica*, v njem pa že ogrodje prvega razreda imenovanega *Class1*. Znotraj tega imenskega prostora bomo napisali nekaj novih razredov.

```
namespace MojaKnjiznica
{
    public class Class1
    {    }
}
```

Prvi razred, ki ga bomo napisali je razred *Oseba*. Ime *Class1* spremenimo v *Oseba* in napišimo še telo razreda.

```
public class Oseba
{
    //zasebna polja razreda Oseba
    private int idOsebe;
    private string ime;
    private string priimek;
    private string kraj;
    private string naslov;
    private int posta;
    //konstruktor
    public Oseba(int id,string ime,string priimek,string kraj,string
                 naslov,int posta)
    {
        idOsebe = id;
        this.ime = ime;
        this.priimek = priimek;
        this.kraj = kraj;
        this.naslov = naslov;
        this.posta = posta;
    }
    //Lastnosti
    public int IdOsebe
    {
        get {return idOsebe;}
        set {idOsebe = value;}
    }
    public string Ime
    {
        get { return ime;}
        set { ime = value;}
```

```
    }
    public string Priimek
    {
        get { return Priimek;}
        set { Priimek = value;}
    }
    public string Naslov
    {
        get { return naslov; }
        set { naslov = value; }
    }
    public string Kraj
    {
        get { return Kraj; }
        set { Kraj = value; }
    }
    public int Posta
    {
        get { return posta; }
        set
        {   //veljavna poštna številka je npr. med 1000 in 10000
            if (value > 1000 && value < 100000)
                posta = value;
            else posta = 0;
        }
    }
 public int Posta
    {
        get { return posta; }
        set { posta = value; }
    }
    /*metoda Izpis je virtualna zato, ker jo bomo v izpeljanem razredu
      prekrili*/
    public virtual string Izpis()
    {
        return ime + " "+ priimek + ", " + naslov + " " + posta + " "+kraj;
    }
}
```

Naslednji razred bo razred *Krvodajalec*, ki podeduje razred *Oseba*. Napišimo ga takoj za razredom *Oseba*. Oba razreda bomo uporabili na koncu tega poglavja, ko bomo izdelali rešitev naloge iz začetka tega poglavja.

```
public class Krvodajalec : Oseba
{
    private string krvnaSkupina;
    private int stDarovanj;
    //tabela možnih osnovnih krvnih skupin
    private string[] skupine = { "A", "B", "AB", "0" };
```

```
    //konstruktor, dedujemo bazični konstruktor
    public Krvodajalec(int idOsebe,string ime,string priimek,string
             kraj,string naslov,int posta,string krvnaSkupina,int stDarovanj)
        :base (idOsebe,ime,priimek,kraj,naslov,posta)
    {
        this.krvnaSkupina = krvnaSkupina;
        this.stDarovanj = stDarovanj;
    }
    //poskrbimo za pravilno nastavitev krvne skupine
    public string KrvnaSkupina
    {
        get { return krvnaSkupina; }
        set {
              //preverimo, če krvna skupina obstaja v seznamu (tabeli)
              if (skupine.Contains(krvnaSkupina))
                  krvnaSkupina=value;
            }
    }
    //poskrbimo za pravilno nastavitev števila darovanj
    public int StDarovanj
    {
        get { return stDarovanj; }
        set {
              if (value>=0)
                stDarovanj = value; ;
            }
    }
    //prepišemo bazično metodo Izpis
    public override string Izpis()
    {
        return base.Izpis()+" "+krvnaSkupina+" "+stDarovanj;
    }
}
```

Dodajmo še razreda *NumberBox* in *Gumb*, ki bosta dedovala vizuelna gradnika *TextBox* in *Button* (*POZOR: v datoteki tipa Class Library lahko dedujemo vizuelne gradnike šele od verzije 2010 naprej!*). Razred *NumberBox* je v bistvu *TextBox*, v katerega pa lahko vnašamo le številke in eno decimalno vejico, vsebuje pa tudi gradnik *ErrorProvider* za preverjanje uporabnikovega vnosa. Razred *Gumb* pa je *Button* s posebnimi oblikovnimi lastnostmi.

V programu bomo uporabili tudi obdelovalec dogodkov *CancelEventHandler*. Njegova naloga je obdelava dogodkov, ki so lahko preklicani. V našem primeru ga potrebujemo za uspešno kontrolo uporabnikovega vnosa.

```
public class NumberBox : TextBox
{
    ErrorProvider eP = new ErrorProvider();
    public NumberBox() //konstruktor
```

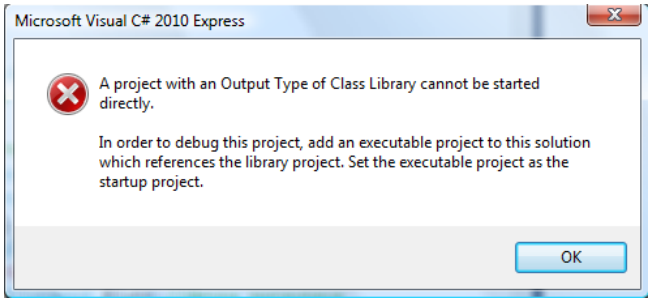
```
    {
        /*Upravljalcu dogodkov KeyPressEventHandler dodajmo dogodek
          KeyPress*/
        this.KeyPress += new KeyPressEventHandler(NumberBox_KeyPress);
        this.BackColor = SystemColors.InactiveBorder;
        this.ForeColor = Color.Navy;
        this.BorderStyle = BorderStyle.FixedSingle;//enojni okvir
        this.Font = new Font("Calibri", 12);
        this.TextAlign = HorizontalAlignment.Right;//desna poravnava
        /*Upravljalcu dogodkov CancelEventHandler dodajmo dogodek
          Validating*/
        this.Validating += new CancelEventHandler(NumberBox_Validating);
    }
    /*dogodek Validating, ki se zgodi, ko se uporabnik premakne na drug
      gradnik*/
    private void NumberBox_Validating(object sender, EventArgs e)
    {
        KontrolaVnosa(); //klic metode
    }
    /*metoda vrne true, če uporabnik vnese vsaj en znak, sicer pa generira
      ustrezno sporočilo*/
    private bool KontrolaVnosa()
    {
        if (this.Text == "")
        {
            eP.SetError(this, "Vsebina polja ne sme biti prazna!");
            return false;
        }
        else
        {
            eP.SetError(this, "");
            return true;
        }
    }
    //vsebina metode ki se izvede ob dogodku KeyPress
    private void NumberBox_KeyPress(object sender, KeyPressEventArgs kpe)
    {
        int KeyCode = (int)kpe.KeyChar;
        /*če vneseni znak NI števka (med 0 in 9) in NISMO stisnili tipke
          BackSpace (koda je 8) in NISMO vnesli decimalne vejice, je dogodek
          zaključen: znaka NE sprejmemo*/
        if (!IsNumberInRange(KeyCode,'0','9')&&KeyCode!= 8 && KeyCode != ',')
        {
            kpe.Handled = true; //dogodek je obdelan!!!
        }
        else
        {
            //dovolimo tudi vnos decimalne vejice
            if (KeyCode == ',')
            {
```

```
                //vnosno polje lahko vsebuje le eno vejico
                kpe.Handled = (this.Text.IndexOf(",") > -1);
            }
        }
    }
    //metoda vrne true, če je vneseni znak med znakoma Min in Max
    private bool IsNumberInRange(int Val, int Min, int Max)
    {
        return (Min <= Val && Val <= Max);
    }
    private void InitializeComponent()
    {
        /*Kadar nastavljamo več atributov nekega gradnika, je pametno
          uporabiti še metodi SuspendLayout in ResumeLayout. Metodi se
          vedno uporabljata v paru, poskrbita pa za pravilno razporeditev
          novih gradnikov na obrazcu, panelu, ipd*/
        this.SuspendLayout();
        this.ResumeLayout(false);
    }
}
```

Naš razred *Gumb* bo enostaven, saj "običajnemu" gumbu (razreda *Button*) spremenimo le konstruktor, v katerem nastavimo določene lastnosti. Si predstavljate, koliko dela bi bilo potrebnega, če dedovanja ne bi poznali in bi hoteli malo drugačen gumb. Napisati bi morali nekaj 100 vrsti kode (povsem podobne tisti za *Button*). In če bi potrebovali še eno vrsto gumbov, bi morali "vajo" s pisanjem kode še enkrat ponoviti, čeprav bi bila koda skoraj identična.

```
public class Gumb : Button
{
    public Gumb() //konstruktor
    {
        this.BackColor =SystemColors.InactiveBorder;
        this.FlatStyle = FlatStyle.Flat;
        this.ForeColor = Color.SteelBlue;
        this.Font = new Font("Verdana", 20, FontStyle.Bold);
        this.Width = 65;
        this.Height = 50;
    }
}
```

Knjižnico moramo sedaj le še prevesti. Kot rezultat prevajanja (če seveda ni sintaktičnih napak) jev mapi *Bin→Debug* znotraj našega projekta tipa *Class Library* je nastala datoteka *MojaKnjiznica.dll* (*dll = Dinamic Link Library*). Če pa skušamo tako nastali projekt že pognati (*F5*), pa dobimo obvestilo



Slika 6: Sporočilno okno, ki se pokaže, če skušamo pognati dll.

Razvojno okolje nas opozori, da prevedeno knjižnico ne moremo uporabiti neposredno, ampak jo lahko le dodamo k nekemu projektu.

Pokazali smo, kako lahko nov vizuelni gradnik izpeljemo iz obstoječega in izpeljavo naredili povsem programsko, s kodo. Obstaja pa še en način. Oglejmo si ga na oprimeru razvoja gradnika, ki bo oznaka, katere napis bo utripal.

V projektu *MojaKnjiznica* v oknu *Solution Explorer* desno kliknimo na *MojaKnjiznica→Add→UserControl*. Odpre se okno *Add New Item*. V oknu izberemo postavko *User Control*, na dnu okna jo poimenujmo *UtripajocaOznaka* in kliknimo *Add*. V oknu *Solution Explorer* se pojavi nov element *UtripajocaOznaka*, v urejevalniku pa se pojavi nov zavihek *UtripajocaOznaka.cs* in v njem podlaga za izdelavo novega gradnika.

Utripanje bomo dosegli tako, da bomo s pomočjo gradnika Timer tej labeli vsako sekundo spreminjali barvo. Na pripravljeno podlago zato postavimo gradnik *Label* in gradnik *Timer*. Gradniku *Label* nastavimo poljuben font in poljubno velikost, gradniku *Timer* pa nastavimo lastnost *Enabled* na *True* in lastnost *Interval* pa na *1000*.

Slika 7: Nov gradnik: utripajoča oznaka.

Gradniku *Timer* zapišimo še ustrezen dogodek *Tick*, obenem pa v konstruktorju novega gradnika zapišimo privzeti font.

```
public partial class UtripajocaOznaka : UserControl
{
    public UtripajocaOznaka()//konstruktor gradnika UtripajocaOznaka
```

```
    {
        InitializeComponent();
        this.Font=new Font("Calibri",18,FontStyle.Bold);//nov privzeti font
    }
    bool spremeni=true;
    private void timer1_Tick(object sender, EventArgs e)
    {
        /*ob vsakem dogodku Tick gradnika Timer se spremeni barva.
          Oznaka zato 'utripa'*/
        if (spremeni)
            label1.ForeColor = Color.LightSteelBlue;
        else
            label1.ForeColor = Color.DarkSlateBlue;
        spremeni = !spremeni;
    }
}
```

V razredu *UtripajocaOznaka* zapišimo še lastnost/propety z imenom *Interval*. S pomočjo te lastnosti bomo tej oznaki spreminjali interval utripanja.

```
public int Interval
{
    get { return timer1.Interval; }
    set { timer1.Interval = value; }
}
```

Še uporabnejši gradnik pa bi lahko pripravili, če bi prek lastnosti omogočili še spreminjanje fonta in para barv, ki se izmenjujeta. Projekt ponovno prevedemo in utripajoča oznaka je pripravljena za uporabo. V nov projekt jo vključimo popolnoma enako kot bomo to storili z gradnikoma *NumberBox* in *Gumb*. Interval utripanja lahko spremenimo kar v oknu *Properties* s pomočjo nove lastnosti *Interval*.

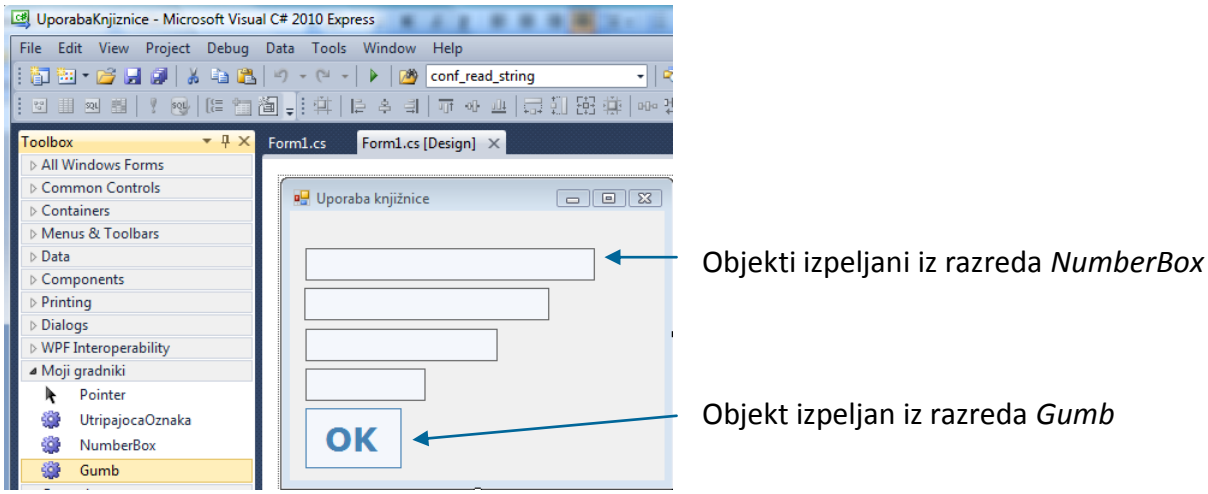
Uporaba lastne knjižnice

Naučiti se moramo še, kako lastno knjižnico uporabimo v novem projektu. Ustvarimo nov projekt in ga poimenujmo *UporabaKnjiznice*. Projektu moramo najprej omogočiti dostop do knjižnice *MojaKnjiznica*. V oknu *Solution Explorer* desno kliknimo na ime projekta in izberimo *Add Reference*. Odpre se okno *Add Reference*, kjer izberemo zavihek *Browse*: v oknu, ki je podoben *Windows Explorer*-ju poiščimo datoteko *MojaKnjiznica.dll*, ki smo jo ustvarili v prejšnji vaji. Izbiro potrdimo s klikom na gumb *OK*. V Oknu *Solution Explorer→References* našega trenutnega projekta se pojavi nova referenca - *MojaKnjiznica*, kar pomeni, da imamo poln dostop do vseh javnih razredov te knjižnice. Do razredov dostopamo preko imena knjižnice, če pa ime knjižnice dodamo v *using* sekcijo, pa je dostop neposreden.

```
using System.Text;
using System.Windows.Forms;
using MojaKnjiznica; //dodana knjižnica
```

```
namespace UporabaKnjiznice
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            /*primer inicializacije objektov ustvarjenih iz razredov v
              datoteki MojaKnjiznica.dll*/
            NumberBox nB1 = new NumberBox();
            Gumb gumb1 = new Gumb();
            Oseba o1 = new Oseba(5092345,"Jan","Kos","Kranj","Kot 3",4000);
            Krvodajalec k1 = new Krvodajalec(3245671,"Anja", "Štrukelj",
"Kranj", "Triglavska 11", 4000, "0", 10);
        }
    }
}
```

V zgornjem primeru smo objekta *nB1* in *gumb1* ustvarili dinamično. Ker pa so razredi *NumberBox*, *Gumb* in *UtripajocaOznaka* razredi, ki smo jih izpeljali iz vizuelnih gradnikov *TextBox*, *Button* in *Label*, lahko nove razrede (nove vizuelne gradnike) postavimo tudi v okno *Toolbox*. To storimo takole: v oknu *Toolbox* za vajo najprej ustvarimo nov zavihek *Moji Gradniki* (desno kliknemo v prazen prostor pod zavihkom *General*, nato izberemo *Add Tab* in kot ime zavihka zapišemo *Moji Gradniki*). Na pravkar ustvarjeni novi zavihek sedaj desno kliknimo in izberimo *Choose Items…* Po nekaj sekundah (lahko pa tudi nekaj minutah, odvisno od zmožnosti računalnika) se pokaže okno *Choose ToolBox Items*. V njem kliknimo gumb *Browse*. Odpre se *Windows Explorer*, s pomočjo katerega poiščimo datoteko *MojaKnjiznica.dll*. Izbiro datoteke potrdimo s klikom na gumb *Open* na dnu okna. Okno *Choose Toolbox Items* še zaprimo s klikom na *OK*. V zavihku *Moji Gradniki* se prikažejo trije novi gradniki *NumberBox*, *Gumb* in *UtripajocaOznaka*, ki jih sedaj lahko uporabljamo tako kot katerikoli drug gradnik.



Slika 8: Nova gradnika *NumberBox* in *Gumb* v oknu *Toolbox*.

Kalkulator

Obrazec *FObrazec*, ki smo ga izdelali na začetku tega poglavja in knjižnico, ki smo jo pravkar izdelali, bomo uporabili za izdelavo lastnega kalkulatorja. Projekt poimenujmo *Kalkulator*. V projekt vključimo našo knjižnico *MojaKnjiznica* (*Solution Explorer*→desni klik na ime projekta→*Add Reference*→*Browse*→poiščimo datoteko *MojaKnjiznica.dll*), nato pa še obrazec *FObrazec*, ki se nahaja v projektu *Obrazec* (v projekt ga dodamo na enak način kor knjižnico, nato pa na začetku datoteke dodamo še stavek *using Obrazec;*). Ta obrazec sedaj dedujemo takole:

```
public partial class Form1 : FObrazec
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

Videz obrazca v pogledu *Design* se zaradi dedovanja spremeni. Nanj postavimo naslednje gradnike: *NumberBox* (*numberBox1* – prikazovalnik števila), dva gumba s sliko tipa *Button* (prvi za prikaz navodil, drugi za prikaz shranjenih števil), vsi ostali gumbi pa so tipa *Gumb* – to je vizuelni gradnik iz knjižnice *MojaKnjiznica*.

Slika 9: Kalkulator.

Na začetku poskrbimo za začetne vrednosti spremenljivk. Prikazovalnik rezultata bo onemogočen več delovanja programa, gumb za izračun pa le na začetku. Ko se obrazec prikaže prvič, aktiviramo prikazovalnik in fokus dodelimo gumbu *plus/minus*.

```
//obrazec Form1 deduje obrazec Obrazec.FObrazec
public partial class Form1 : Obrazec.FObrazec
{
    double steviloVSpominu=0; //število, ki ga shranjujemo v spomin
    //število ki mu prištevamo/odštevamo oz. ga množimo/delimo
    double prvoStevilo;
    char operacija;//vrsta operacije

    public Form1() //konstruktor obrazca
    {
        InitializeComponent();
        gEnako.Enabled = false;
        numberBox1.Enabled = false; //skrijemo kurzor v vnosnem polju
    }

    /*na začetku mora biti izbran prikazovalnik, fokus pa ima gumb plus-
      minus*/
    private void Form1_Shown(object sender, EventArgs e)
    {
        numberBox1.Select();
        gPlusMinus.Focus();
    }
}
```

Uporabniku omogočimo brisanje prikazanih števk v prikazovalniku. Odzivno metodo dogodka *Click* priredimo gumbu z oznako '<'. Vsak klik na ta gumb pomeni brisanje najbolj desne števke v prikazovalniku.

```
//brisanje najbolj desne števke prikazovalnika
private void gBrisi_Click(object sender, EventArgs e)
{
    try
    {
        /*če je v prikazovalniku več kot ena števka, odstranimo najbolj
          desno števko*/
        if (numberBox1.Text.Length > 0)
            numberBox1.Text = numberBox1.Text.Substring(0,
                 numberBox1.Text.Length - 1);
        /*če je prikazovalnik prazen, ali pa je v njem le predznak oz
          znaka '-0' vanj zapišemo ničlo */
        if (numberBox1.Text.Length == 0 || numberBox1.Text == "-" ||
              numberBox1.Text == "-0")
            numberBox1.Text = "0";
    }
    catch { }
```

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

```
}
```

Vsem gumbom na obrazcu, razen gumboma s sliko, priredimo odzivno metodo dogodka *KeyPress*, vsakemu posebej pa bomo morali napisati še odzivno metodo dogodka *Click*. Kalkulator bomo namreč lahko upravljali s tipkovnico ali pa z klikanjem miške. Ob pritisku numerične tipke na tipkovnici, ali pa kliku miške na nek gumb s številko, se bo v prikazovalniku prikazala ustrezna številka. Pri tem pa moramo paziti, da pri večkratni zaporedni uporabi tipke z decimalno vejico le-ta ne bo prikazana dvakrat. V ta namen bomo napisali metodo *gVejica_Click*. Stisk tipke '<' požene metodo za brisanje najbolj desnega znaka v prikazovalniku. Tipka 'c' ali pa 'C' ima za nalogo brisanje prikazovalnika. Če uporabnik pritisne tipko '+', '-', '*' in '/', gumbe s temi znaki začasno onemogočimo, da jih ne more pritisniti večkrat zapored, operacijo pa si zapomnimo za kasnejši izračun. Dodamo še klic odzivne metode *gEnako_Click*, ki se izvede če uporabnik stisne ali pa klikne gumb z enačajem.

```
//pritisk tipke na tipkovnici
private void Gumbi_KeyPress(object sender, KeyPressEventArgs e)
{
    /*če je na tipkovnici kliknjena tipka od 0 - 9, jo dodamo v
      numberBox1*/
    if ((e.KeyChar >= '0') && (e.KeyChar <= '9'))
    {
        //če je v oknu le številka 0, jo prej odstranimo
        if (numberBox1.Text == "0" || numberBox1.Text=="-0")
            numberBox1.Clear();
        numberBox1.Text += e.KeyChar;//dodam vtipkano številko
    }
    else if (e.KeyChar == ',')
    {
        gVejica_Click(sender, e);//dovoljena je le ena vejica
    }
    else if ((e.KeyChar == (char)8))//tipka BackSpace
    //ali pa takole: else if (e.KeyChar ==Convert.ToChar(Keys.Back))
        gBrisi_Click(sender, e);
    else if ((e.KeyChar == 'c' || e.KeyChar == 'C'))
        gC_Click(sender, e);
    else if (e.KeyChar == '+' || e.KeyChar == '-' || e.KeyChar == '*' ||
             e.KeyChar == '/')
    {
        prvoStevilo = Convert.ToDouble(numberBox1.Text);
        numberBox1.Text = "0";//brišemo vsebino prikazovalnika
        Operacije(false); //začasno onemogočimo tipke za mat. operacije
        operacija = e.KeyChar; //operacijo si zapomnimo
    }
    //če stisnjena tipka '=' za izračun in je gumb '=' omogočen
    else if (e.KeyChar == '=' && gEnako.Enabled )
        gEnako_Click(sender, e);
    else { }
}
```

Napisati moramo še odzivne metode za dogodke *Click* vseh gumbov na obrazcu. Za gumbe s številkami je odzivna metoda skupna, ostalim gumbom pa pripadajo svoje metode. Imena metod so taka, da ne bo težko ugotoviti, kateremu gumbu pripadajo.

```
//klik na gumb s številko
private void Stevka_Click(object sender, EventArgs e)
{
    //če je trenutna vsebina enaka 0, jo pobrišem
    if (numberBox1.Text == "0"||numberBox1.Text == "-0")
        numberBox1.Clear();
    //v numberBox dodam pritisnjeno številko
    numberBox1.Text += (sender as Button).Text;
}

//metoda gPlus_Click se izvede ob pritisku na tipke '+','-','*' ali '/'
private void gPlus_Click(object sender, EventArgs e)
{
    prvoStevilo = Convert.ToDouble(numberBox1.Text);
    numberBox1.Text = "0";
    Operacije(false); //onemogočim tipke '+','-','*' ali '/'
    //zapomnimo si vrsto operacije
    operacija = Convert.ToChar((sender as Button).Text);
}

//klik na gumb plus-minus
private void gPlusMinus_Click(object sender, EventArgs e)
{
    try
    {
        //zamenjava predznaka
        double st = Convert.ToDouble(numberBox1.Text);
        //če je predznak minus, ga odstranimo
        if (numberBox1.Text[0]=='-')
            numberBox1.Text = numberBox1.Text.Substring(1,
                numberBox1.Text.Length-1);
        else //sicer pa ga dodamo
            numberBox1.Text = '-' + numberBox1.Text;
    }
    catch { }
}

//klik na gumb z decimalno vejico
private void gVejica_Click(object sender, EventArgs e)
{
    //vejico dodamo le, če je še ni!
    if (!numberBox1.Text.Contains(','))
        numberBox1.Text += ',';
}
```

```
//klik na gumb Shrani
private void gShrani_Click(object sender, EventArgs e)
{
    /*trenutno vsebino gradnika numberBox shranimo v spremenljivko stevilka*/
    try
    {
        steviloVSpominu = Convert.ToDouble(numberBox1.Text);
    }
    catch { }
}

//klik na gumb Prikaži
private void gPrikazi_Click(object sender, EventArgs e)
{
    //shranjeno vrednost spremenljivke v spominu zapišemo v numberBox
    numberBox1.Text = steviloVSpominu.ToString();
}
//klik na gumb C
private void gC_Click(object sender, EventArgs e)
{
    //brišemo vsebino
    numberBox1.Text = "0";
}
//klik na gumb Ce = začetno stanje kalkulatorja
private void gCe_Click(object sender, EventArgs e)
{
    numberBox1.Text = "0";
    steviloVSpominu = 0;
    Operacije(true); //omogočimo tipke '+','-','*' ali '/'
}

//klik na gumb za izračun rezultata
private void gEnako_Click(object sender, EventArgs e)
{
    //zapomnimo si drugo vneseno stevilo
    double drugoStevilo = Convert.ToDouble(numberBox1.Text);
    double rezultat=0;
    //izračun rezultata
    switch (operacija)
    {
        case '+': rezultat = prvoStevilo + drugoStevilo;
            break;
        case '-': rezultat = prvoStevilo - drugoStevilo;
            break;
        case '*': rezultat = prvoStevilo * drugoStevilo;
            break;
        case '/': rezultat = prvoStevilo / drugoStevilo;
            break;
    }
    //izpis rezultata na prikazovalniku
```

```
    numberBox1.Text = rezultat.ToString();
    //omogočim tipke '+','-','*' ali '/' in onemogočim tiko '='
    Operacije(true);
}

//onemogočanje/omogočanje gumbov
private void Operacije(bool log)
{
    gPlus.Enabled = log;
    gMinus.Enabled = log;
    gKrat.Enabled = log;
    gDeljeno.Enabled = log;
    gEnako.Enabled = !log;
}

//izpis navodil
private void button1_Click(object sender, EventArgs e)
{
    string navodila = "Uporabljaš lahko miško ali pa tipkovnico!\n\n";
    navodila += "Ko vnašaš številko, lahko z gumbom '<' brišeš zadnje
                 vnesene znake.\n";
    navodila += "Ob kliku na gumbe '+', '-', '*' ali '/' se številka na
              prikazovalniku shrani v delovni pomnilnik za računanje.\n";
    navodila += "Trenutno številko na prikazovalniku pobrišemo z gumbom
                'C'.\n";
    navodila += "Klik na gumb 'Ce' pobriše številko na prikazovalniku in
                 številko v spominu.\n";
    navodila += "Klik na gumb 'Shrani' številko na prikazovalniku shrani
                 za kasnejšo uporabo.\n";
    navodila += "Klik na gumb 'Prikaži' na prikazovalniku prikaže
                 shranjeno številko.\n";
    navodila += "Gumb '=' je neaktiven vse dokler ne pritisnemo enega od
                 gumbov '+', '-', '*' ali '/'.\n";
    navodila += "\nPostopek pri delu s tipkovnico:\n1.)Vtipkaj prvo
                 število;";
    navodila += "\n2.)Klikni gumb '+', '-', '*' ali '/';";
    navodila += "\n3.)Vtipkaj drugo število;\n4.)Klikni gumb '=' za izpis
                 rezultata na prikazovalniku!";
    navodila += "\n\nKo se na prikazovalniku prikaže rezultat, imaš na
                 voljo 3 možnosti:";
    navodila += "\n  - klikneš gumba 'C' ali 'Ce' za brisanje številke;";
    navodila += "\n  -  k rezultatu dodajaš nove števke in računaš
                 naprej;";
    navodila += "\n  -  vneseš poljubno operacijo '+', '-', '*' ali '/'
                 in nato vneseš novo številko!";
    MessageBox.Show(navodila, "Navodila za
                               uporabo",0,MessageBoxIcon.Information);
}

//prikaz števila v delovnem pomnilniku in spominu
```

```
private void button2_Click(object sender, EventArgs e)
{
    string pomnilnik = "Število v delovnem pomnilniku:
                       "+prvoStevilo+"\n\n";
    pomnilnik += "Število v spominu: " + steviloVSpominu;
    MessageBox.Show(pomnilnik, "Številki v delovnem pomnilniku in
       pomnilniku",0,MessageBoxIcon.Exclamation);
}
```

Abstraktni razredi in abstraktne metode

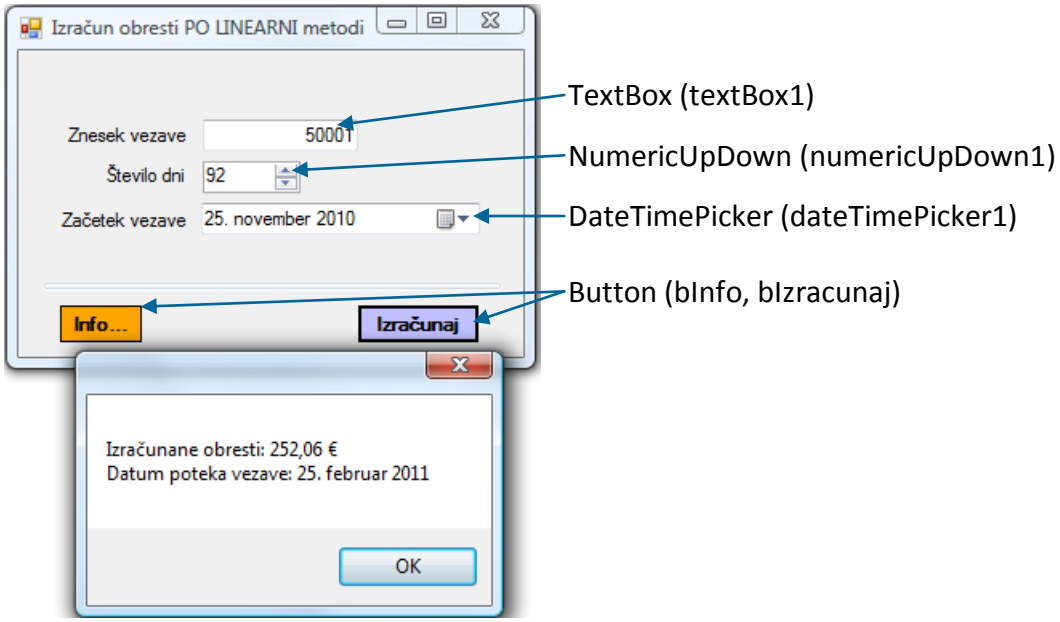
Abstraktni razredi so razredi, iz katerih ne moremo tvoriti objektov, ampak jih lahko le dedujemo, oziroma tvorimo izpeljane razrede. Namen takih razredov je, da poskrbijo za temeljno definicijo bazičnega razreda, uporaba le-tega pa bo implementirana v številnih izpeljanih razredih. Razred za abstraktnega proglasimo s pomočjo rezervirane besede *abstract*.

V abstraktnih razredih se pogosto pojavijo tudi *abstraktne metode*. Pred tipom take metode zapišemo besedico *abstract*. Abstraktne metode imajo samo glavo, ki ji sledi podpičje, nimajo pa implementacije (nimajo telesa metode). Razredi, ki jih izpeljemo iz abstraktnih razredov morajo zato implementirati vse abstraktne metode.

```
abstract class abstraktniRazred
{
    //telo abstraktnega razreda
    //najava abstraktne metode
    public abstract void metoda();
}

//dedujemo abstraktni razred
public class izpeljaniRazred : abstraktniRazred
{
    //telo izpeljanega razreda
    public override void metoda()
    {
       //implementacija abstraktne metode
    }
}
```

Abstraktni razred je lahko izpeljan tudi iz nekega osnovnega razreda.

Kot primer abstraktnega razreda napišimo abstraktni razred *Racun*, ki predstavlja abstraktno osnovo transakcijskega računa vseh komitentov neke banke. Razred nato implementirajmo v projekt, kjer bomo abstraktni razred *Racun* dedovali v razredu *Depozit*. Projekt bo vseboval en sam obrazec za vnos potrebnih podatkov za izračun obresti po linearni metodi! Vizuelno bo obrazec izgledal takole:



Slika 10: Obrazec za izračun obresti.

Abstraktni razred naj vsebuje podatek o imenu komitenta in stanju na računu, ter ustrezen konstruktor. Dodajmo mu še objektno metodo *Transakcija*, ki bo skrbela za ažuriranje stna računa. Predpostavimo, da bomo v izpeljanem razredu potrebovali metodo *Obresti* za izračun obresti. V našem abstraktnem razredu jo zato napovejmo kot abstraktno metodo.

```
public partial class Form1 : Form
{
    //Abstraktni razred: iz njega ne moremo neposredno tvoriti objektov
    public abstract class Racun
    {
        public string ime; //polje dostopno v bazičnem in izpeljanem razredu
        public double stanje;//polje dostopno v bazičnem in izpelj. razredu
        public Racun(string ime, double stanje) //bazični konstruktor
        {
            this.ime = ime;
            this.stanje = stanje;
        }
        //metoda za izračun novega stanja na računu, glede na polog oz. dvig
        public void Transakcija(double znes)
        {
            this.stanje += znes;
        }
        //napoved abstraktne metode: implementirana bo v izpeljanem razredu
        public abstract double Obresti();//
    }
}
```

Razred *Depozit* deduje abstraktni razred *Racun*. Dodajmo mu še tri polja, potrebna za hranjenje podatkov o vezavi sredstev in pripadajoči konstruktor. Le-ta naj deduje konstruktor abstraktnega razreda. Napišimo tudi objektno prekrivno metodo *Obresti*, ki smo jo napovedali že prej. Za potrebe izpisa dodamo še metodo *ToString*, ki bo prekrila bazično metodo z istim imenom. Metoda bo vračala podatke o komitentu.

```
//Razred Depozit deduje razred Racun
public class Depozit : Racun
{
    public double znesek;
    public int dni;
    public DateTime zacetekVezave;
    //Konstruktor
    public Depozit(string ime, double stanje, double znesek, int dni,
DateTime datum): base(ime, stanje) //Podedujemo konstruktor bazičnega razreda
    {
        this.znesek = znesek;
        this.dni = dni;
        this.zacetekVezave = datum;
    }

    /*metoda Obresti implementira abstraktno metodo, izračuna pa obresti
      PO LINEARNI metodi!!!*/
    public override double Obresti()
    {
        //višina obresti je odvisna od zneska vezave
        double procent = obresti1;
        if (znesek>10000 && znesek<=50000)
            procent = obresti2;
        else if (znesek>50000)
            procent = obresti3;
        //izračun obresti
        double obresti=Math.Round(znesek*(double)procent/100*dni/365, 2);
        return obresti;
    }
    //metoda ToString() prekrije javno metodo ToString()
    public override string ToString()
    {
        return "Komitent: " + ime + ", znesek vezave: " + znesek + ", število
dni vezave: " + dni + ", začetek vezave: " + zacetekVezave + ",  pripadajoče
obresti: " + Obresti();
    }
}
```

Veljavne obrestne mere bomo zaradi enostavnosti kar določili, lahko pa bi jih imeli shranjene v neki datoteki, ki bi jo v programu ustrezno ažurirali. Obema gumboma na obrazcu napišimo še odzivni metodi dogodka *Click*.

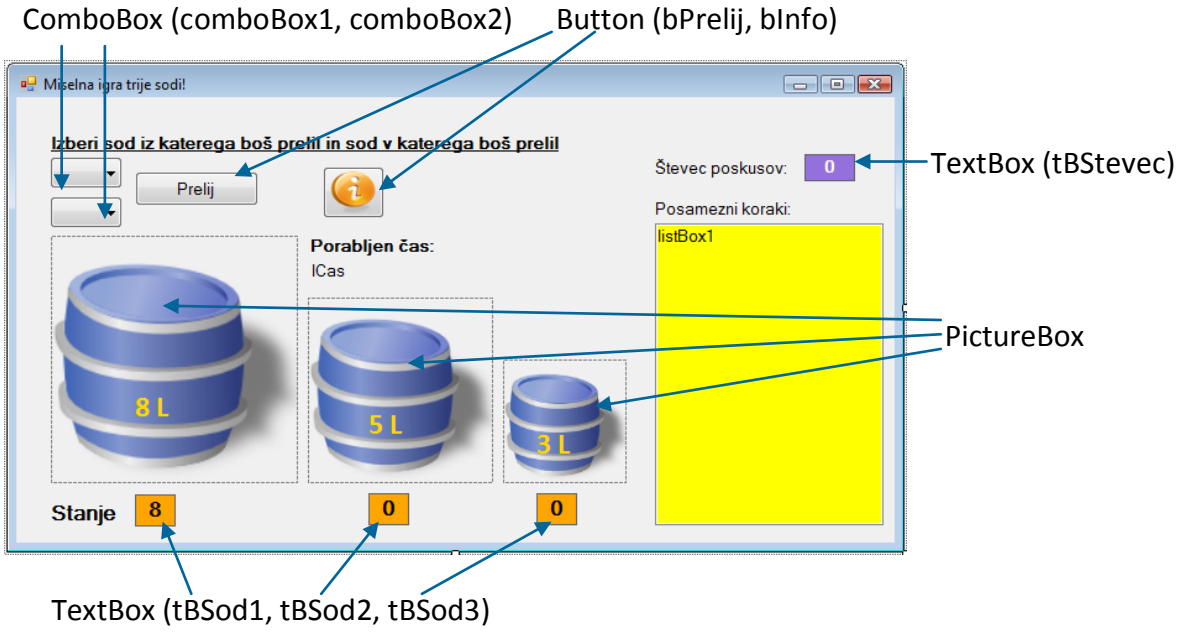
```
//trenutne obresti so konstante
```

```
const double obresti1 = 1.90; //obresti za zneske do 10000 EUR
const double obresti2 = 1.95; //obresti za zneske od 10001 do 50000 EUR
const double obresti3 = 2.00; //obresti za zneske nad 50000 EUR

public Form1()
{
    InitializeComponent();
}

private void bIzracunaj_Click(object sender, EventArgs e)
{
    try
    {
        double znesek = Convert.ToDouble(textBox2.Text);
        int dni = Convert.ToInt32(numericUpDown1.Value);
        /*ustvarjanje novega objekta – konstruktorju posredujemo število dni
in znesek vezave/kredita. Ime komitenta in trenutno stanja računa nas ne
zanimata, zato vstavimo vrednosti "" in  0*/
        Depozit rc1 = new Depozit("", 0, znesek,dni,dateTimePicker1.Value);
        DateTime datumKoncaVezave =
dateTimePicker1.Value.AddDays((int)numericUpDown1.Value);
        MessageBox.Show("Izračunane obresti: "+rc1.Obresti().ToString()+"
€\nDatum poteka vezave: "+datumKoncaVezave.ToLongDateString());
    }
    catch
    {
        MessageBox.Show("Napaka v podatkih!");
    }
}

private void bInfo_Click(object sender, EventArgs e)
{
    MessageBox.Show("Trenutna obrestna mera:\n\nZneski do 10.000,00 €:
"+obresti1+" %\nZneski med 10.001,00 € in 50.000,00 €:  "+obresti2+"
%\nZneski nad 50.000,00 €:  "+obresti3+" %");
}
```

Izračunane obresti smo prikazali v sporočilnem oknu kar s pomočjo naše lastne objektne prekrivne metode *ToString*.

Trije sodi

Trije sodi je miselna igra, pri kateri imamo tri sode, ki držijo 8 litrov, 5 litrov in 3 litre. Prvi sod je poln. V čim manj korakih je potrebno s prelivanjem doseči, da bo prvih dveh sodih natanko 4 litre tekočine.



Slika 11: Miselna igra trije sodi!

Za potrebe projekta definirajmo abstraktni razred V*alj* s poljema polmer in višina valja, konstruktorjem ter abstraktno metodo *Kapaciteta*. Iz razreda *Valj* bomo izpeljali razred *Sod*, ki mu dodamo še polje *stanje*, ter virtualno metodo *Kapaciteta*. V oba gradnika *ComboBox* dodamo tri vrstice (*sod1*, *sod2* in *sod3*), lastnost *DropDown* pa nastavimo na *DropDownList*. Uporabnik mora s pomočjo gradnikov *ComboBox* izbrati sod iz katerega bo prelival in sod v katerega bo prelival, izbiro pa potrditi s klikom na gumb *Prelij*. Stanje posameznega soda je zapisano v gradnikih *tBSod1*, *tBSod2* in *tBSod3* pod posameznim sodom, posamezno prelivanje je zapisano še v gradniku *listBox1*, v gradniku *tBStevec* pa je zapisano tudi dosedanje število poskusov. Igra se zaključi, ko je stanje prvega in drugega soda enako 4 litre. Vsi gradniki tipa *TextBox* imajo lastnost *ReadOnly* nastavljeno na *True*.

Recimo, da ne poznamo kapacitete sodov, ampak le polmere in višine. Polmeri vseh treh sodov so enaki 1, višine pa zaporedoma *8 / Math.PI*, *5 / Math.PI* in *3 / Math.PI*. Za izračun prostornine posameznega soda bomo vzeli kar matematično formulo za izračun prostornine valja (*Math.PI * polmer * polmer * visina*). Program bo zato možno kasneje nadgraditi tako, da bo polmere in višine sodov določil kar uporabnik sam. Prav gotovo bi se dalo nalogo rešiti brez uporabe (abstraktnih) razredov, a gre za vajo, katere namen je spoznavanje novega pojma na konkretnem primeru.

Najprej napiimo oba razreda. V razredu *Valj* je metoda *Kapaciteta* abstraktna, kar pomeni, da jo bomo napisali v izpeljanem razredu *Sod*.

```
public partial class Form1 : Form
{
    //abstraktni razred Valj
```

```
    public abstract class Valj
    {
        public double polmer, visina;
        public Valj(double polmer, double visina)
        {
            this.polmer = polmer;
            this.visina = visina;
        }
        //abstraktna metoda za kapaciteto/prostornino valja
        public abstract double Kapaciteta();
    }
    //razred Sod izpeljemo iz abstraktnega razreda Valj
    class Sod:Valj
    {
        public int stanje; //trenutno stanje soda
        public Sod(double polmer, double visina,int stanje) //konstruktor
            :base(polmer,visina)
        {
            this.stanje = stanje;
        }
        public override double Kapaciteta()
        {
            return Math.PI * polmer * polmer * visina;
        }
    }

    static Sod[] sodi = new Sod[3]; //napoved tabele treh sodov
    //spremenljivka, v katero bomo shranili začetni čas igre
    static int stevec = 0;  //števec ptelivanj
    static DateTime zacetek;
    static TimeSpan cas;     //spremenljivka, ki bo hranila porabljen čas
    private void timer1_Tick(object sender, EventArgs e)
    {
        cas = DateTime.Now-zacetek;  //izračunamo porabljen čas
        lCas.Text = cas.ToString(); //osvežimo porabljen čas na oznaki
    }
    private void bInfo_Click(object sender, EventArgs e)
    {
        string navodila = "NAVODILA ZA IGRO\n\nV prvem sodu je 8 litrov
tekočine, druga dva soda pa sta prazna!\n\nV čim manj korakih moraš s
prelivanjem doseči, da bo v prvih dveh sodih natanko 4 litre tekočine!";
       MessageBox.Show(navodila,"NAVODILA",MessageBoxButtons.OK,
                       MessageBoxIcon.Information);
    }
}
```

Napovedali smo tudi tabelo treh sodov, to je objektov izpeljanih iz razreda *Sod*. Merili bomo tudi število prelivanj in čas, potreben za uspešno prelivanje. Porabljeni čas bomo ves čas prelivanja prikazovali na labeli *lCas*. V ta namen smo napisali odzivno metodo dogodka Tick

gradnika *Timer*, ki smo ga pred tem postavili na obrazec. Dodali smo še odzivno metodo gumba *bInfo*, kjer v sporočilnem oknu izpišemo navodila za igro.

Preden se obrazec prvič prikaže poskrbimo za inicializacijo tabele treh sodov. S pomočjo metode *Kapaciteta* izračunamo njihovo prostornino.

```
public Form1()
{
    InitializeComponent();
    //inicializacija tabele sodov
    /*za višine sodov uporabimo take vrednosti, da bodo kapacitete
      sodov 8 litrov, 5 litrov in 3 litre*/
    //prvi sod ima kapaciteto 8 litrov in je na začetku poln
    sodi[0] = new Sod(1, 8 / Math.PI, 8);
    tBSod1.Text = sodi[0].stanje.ToString();
    lSod1.Text = ((int)sodi[0].Kapaciteta()).ToString()+" lit";
    //drugi sod ima kapaciteto 5 litrov in je na začetku prazen
    sodi[1] = new Sod(1, 5 / Math.PI, 0);
    lSod2.Text = ((int)sodi[1].Kapaciteta()).ToString() + " lit";
    //tretji sod ima kapaciteto 3 litre in je na začetku prazen
    sodi[2] = new Sod(1, 3 / Math.PI, 0);
    lSod3.Text = ((int)sodi[2].Kapaciteta()).ToString() + " lit";
    label8.Visible = false;
    lCas.Visible = false;
}
```

Najpomembnejša odzivna metoda programa je metoda gumba *bPrelij*. Preveriti moramo, katera soda je uporabnik izbral in ali je prelivanje sploh možno.

```
private void bPrelij_Click(object sender, EventArgs e)
{
    if (stevec == 0) //če smo na začetku začnemo meriti čas
    {
        label8.Visible = true;//labela z napisom Porabljen čas
        lCas.Visible = true;  //prikažemo labelo s tekočim časom
        zacetek = DateTime.Now; //začnemo meriti čas
    }
    if (comboBox1.Text =="")
        MessageBox.Show("Izberi sod iz katerega boš prelival!", "POZOR!",
MessageBoxButtons.OK, MessageBoxIcon.Information);
    else if (comboBox2.Text=="")
        MessageBox.Show("Izberi sod v katerega boš prelival!", "POZOR!",
MessageBoxButtons.OK, MessageBoxIcon.Information);
    else if (comboBox1.Text == comboBox2.Text)
        MessageBox.Show("Ne moreš prelivati v isti sod.\nIzbira sodov mora
biti različna", "POZOR!", MessageBoxButtons.OK, MessageBoxIcon.Information);
    else if (sodi[comboBox1.SelectedIndex].stanje==0)
```

```
        MessageBox.Show("Sod "+comboBox1.Text+", ki ga želiš preliti, je
prazen!", "POZOR!", MessageBoxButtons.OK, MessageBoxIcon.Error);
    else     //preverimo, če je pretakanje sploh možno
    {
        /*metodi pošljemo indeksa obeh izbir v gradnikih ComboBox, to
          pa sta obenem indeksa obeh sodov v tabeli sodi*/
        Pretoci(comboBox1.SelectedIndex, comboBox2.SelectedIndex);
    }
}
```

Za prelivanje smo v zgornji metdi klicali metodo *Pretoci*, ki jo moramo sedaj napisati. Metoda dobi za parametra številki obeh sodov, rezultat metode pa je novo stanje tekočine v teh dveh sodih. Po prelivanju metoda tudi preveri, če igra že končana. V tem primeru izpiše ustrezno sporočilo, v kateren uporabnika obvesti o številu porabljenih prelivanj. Na obrazcu se zaustavi tudi tekoči čas.

```
private void Pretoci(int prvi, int drugi)
{
    if (sodi[drugi].Kapaciteta() == sodi[drugi].stanje)
        MessageBox.Show("Sod številka 2 je poln, prelivanje ni možno!",
"POZOR!", MessageBoxButtons.OK, MessageBoxIcon.Information);
    else
    {
       /*preverimo, če je v sodu, v katerega prelivamo, dovolj
         prostora za izlitje vse tekočine iz prvega soda */
       if (sodi[prvi].stanje >= sodi[drugi].Kapaciteta() -
sodi[drugi].stanje)
        {
            sodi[prvi].stanje = sodi[prvi].stanje -
((int)sodi[drugi].Kapaciteta() - sodi[drugi].stanje);
            sodi[drugi].stanje = sodi[drugi].stanje +
((int)sodi[drugi].Kapaciteta() - sodi[drugi].stanje);
        }
        else
        {   //v drugi sod lahko izlijemo celotno vsebino prvega soda
            sodi[drugi].stanje = sodi[drugi].stanje + sodi[prvi].stanje;
            sodi[prvi].stanje = 0;
        }
    }

    stevec++; //povečamo števec poskusov
    tBSod1.Text = sodi[0].stanje.ToString(); //stanje prvega soda
    tBSod2.Text = sodi[1].stanje.ToString(); //stanje drugega soda
    tBSod3.Text = sodi[2].stanje.ToString(); //stanje tretjega soda
    listBox1.Items.Add(comboBox1.Text + " -> " + comboBox2.Text + ", stanje:
" + tBSod1.Text + ",  " + tBSod2.Text + ",  " + tBSod3.Text+"   ");
    tBStevec.Text = stevec.ToString(); //ažuriramo prikazani števec poskusov
    //spustna seznama postavimo v začetno stanje
```

```
    comboBox1.SelectedIndex = -1;
    comboBox2.SelectedIndex = -1;
    if (sodi[0].stanje == 4 && sodi[1].stanje == 4)
    {
        timer1.Enabled = false;  //ustavimo Timer (merjenje časa)
        //onemogočimo gumb Prelij - igra je končana
        bPrelij.Enabled = false;
        //onemogočimo izbiro v prvem spustnem seznamu - igra je končana
        comboBox1.Enabled = false;
         //onemogočimo izbiro v drugem spustnem seznamu - igra končana
        comboBox2.Enabled = false;
        MessageBox.Show("Čestitam, uspelo ti je v " + stevec + " poskusih!",
"IGRA KONČANA", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);
    }
}
```

Abstraktni razredi in virtualne metode

Abstraktni razred lahko deduje tudi virtualno metodo iz nekega osnovnega razreda. V tem primeru mora biti prekrivna metoda abstraktna, npr:

```
//osnovni razred
public class Osnovni
{
    public virtual void Nekaj(int i)
    {
        // Originalna implementacija
    }
}

//abstraktni razred lahko podeduje osnovni razred
public abstract class Abstraktni : Osnovni
{
    public abstract override void Nekaj(int i); //abstraktna metoda
}

//razred F izpeljemo iz abstraktnega razreda
public class F : Abstraktni
{
    //prekrivanje osnovne metode Nekaj osnovnega razreda Osnovni
    public override void Nekaj(int i)
    {
        // Nova implementacija
    }
}
```

Če je neka virtualna metoda deklarirana kot abstraktna, je še vedno virtualna v vseh razredih, ki dedujejo abstraktni razred. Razred, ki podeduje abstraktni razred torej nima dostopa do originalne implementacije take metode – v zgornjem primeru torej metoda *Nekaj* razreda *F* **NE** more klicati metode *Nekaj* razreda *Osnovnii*.

Večokenske aplikacije

Ob izdelavi novega projekta razvojno okolje *Visual C#* odpre en sam obrazec, ki ga poimenuje *Form1*. V vseh dosedanjih primerih in vajah smo na tako pripravljen obrazec postavljali gradnike, napisali odzivne metode za dogodke in pisali svoje metode. Slej ko prej pa se pojavi potreba po izdelavi projekta z več obrazci oz. po večokenski aplikaciji.

Preden pa pokažemo, kako izdelamo dodatne obrazce in kako jih odpiramo, pa je potrebno pojasnilo o tem, kakšne vrste obrazcev poznamo. Vsi obrazci so pravzaprav istega tipa, le odpiramo jih lahko na dva načina. Način odpiranja obrazca načeloma spremeni celoten način obdelave rezultatov tega obrazca. Zato ločimo dve vrsti obrazcev:

- *Pogovorna okna* oz. *Dialogi*: to so obrazci, ki jih prikažemo uporabniku in mu ponudimo določene vnose ali pa ga le vprašamo za neko odločitev. Ko uporabnik obrazec zapre, le-ta vrne vrednost na osnovi katere poteka nadaljnje izvajanje aplikacije. Takim obrazcem pravimo tudi dialogi oz. *modalni* obrazci. Odpremo jih z metodo *ShowDialog*. Metoda *ShowDialog* vrne način zapiranja obrazca. Bistvo modalnih obrazcev je tudi v tem, da vsem ostalim oknom znotraj aplikacije preprečujejo, da bi postala aktivna (prejela fokus) dokler je modalni obrazec odprt. Modalni obrazci so tako idealni v situacijah, ko ne bi imelo nobenega smisla, da se program nadaljuje oz. da se zgodi karkoli, dokler uporabnik ne odgovori na določena vprašanja, ali pa dokler ne vnese ustreznih podatkov v vnosna polja.

- *Nemodalni* obrazci: odpremo jih z metodo *Show*. Taki obrazci dovoljujejo uporabniku nadaljevanje dela v drugem obrazcu, ne da bi prej zaprl obrazec, odprt z metodo *Show*. Klasičen primer tako odprtega podobrazca je npr. okno *Find And Replace* v *Microsoft Word*-u. Uporabnik lahko vnese v okno iskalni niz (besedo) in besedo za njegovo zamenjavo, ter nato klikne ustrezen gumb za iskanje in zamenjavo na dnu okna. Seveda pa se lahko uporabnik premisli in nadaljuje z delom v urejevalniku ne da bi mu bilo potrebno okno za iskanje zapreti. Okno za iskanje podatka je izgubilo fokus ne da bi se zaprlo – ostane torej odprto. Uporabnik se lahko kadarkoli vrne v to okno.

Nemodalno odprti obrazci ne zahtevajo posebne obravnave. Obrazec moramo seveda vnaprej pripraviti (ga ustvariti, nanj postaviti ustrezne gradnike,…), nato pa le pravilno zapisati ustrezno

kodo za njegovo odpiranje. Modalno odprti obrazci pa zahtevajo poseben način obravnave o tem, kako jih odpreti, kaj taki obrazci vračajo in kaj storiti, ko jih zapremo.

Izdelava novega obrazca

V že odprtem projektu lahko naredimo nov obrazec in ga s tem dodamo v trenutni projekt na dva načina:

- v meniju *Project* izberemo opcijo *Add Windows Form…* Prikaže se pogovorno okno, v katerem je že izbrana opcija *Windows Form*, na dnu tega okna pa nam *Visual C#* predlaga ime datoteke, na kateri bo koda novega obrazca (npr. *Form2.cs*). Ime lahko seveda spremenimo (le končnica mora biti *.cs*) in vnos potrdimo s klikom na gumb *Add*. V oknu *Solution Explorer* se pojavi nova postavka za pravkar ustvarjeni obrazec – ta obrazec postane tudi aktiven in se prikaže v urejevalniškem oknu.

- V oknu *Solution Explorer* desno kliknimo na trenutni projekt, izberemo *Add→Windows Form…* Prikaže se pogovorno okno, v katerem je tako kot pri prvem načinu že izbrana postavka *Windows Form*. S klikom na *Add* potrdimo (oz. spremenimo) ime novega obrazca.

Na tako ustvarjen nov obrazec lahko postavljamo gradnike in pišemo odzivne metode tako kot pri osnovnem obrazcu. Na ta način lahko izdelamo poljubno število novih obrazcev. Med njimi pa preklapljamo (da jih lahko urejamo in nanje postavljamo nove gradnike) z dvoklikom na ime obrazca v oknu *Solution Explorer*.

Odpiranje nemodalnih obrazcev

Visual C# ob zagonu programa odpre samo en obrazec. Če projekt vsebuje več obrazcev, se bo na začetku prikazal obrazec, katerega ime je zapisano v datoteki *Program.cs*.

```
static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        //Najprej se bo odprl obrazec z imenom Form1
        Application.Run(new Form1());
    }
}
```

V zgornjem primeru se bo torej najprej odprl obrazec *Form1*. Če želimo, da se namesto tega obrazca odpre nek drug, samo popravimo ime obrazca.

Odpiranje drugih obrazcev je potrebno zagotoviti programsko, s kodo, ne glede nato, ali odpiramo modalen ali nemodalen obrazec. Pravzaprav pa smo o tem govorili že velikokrat, ko smo odpirali obrazec *MessageBox*. Razlika je le v tem, da je *MessageBox* obrazec, ki je že sestavni del razvojnega okolja, mi pa bomo ustvarjali svoje obrazce. Odpiranje obrazcev bomo torej običajno vezali na dogodek *Click* poljubnega gradnika (npr. gumba, vrstice v gradniku *DataGridView*, gradnika *PictureBox*, ipd.) ali pa npr. postavko poljubnega menija. Nemodalni obrazec odpremo z metodo *Show* takole:

```
//Odpiranje nemodalnega obrazca ob kliku na gumb z imenom bOdpri
private void bOdpri_Click(object sender, EventArgs e)
{
    //izdelava novega objekta izpeljanega iz obrazca Form2
    Form2 imeObrazca = new Form2();
    imeObrazca.Show();//nemodalno odpiranje obrazca izpeljanega iz Form2
}
```

Nemodalni obrazec lahko zapremo na dva načina. Klasičen način je s pomočjo sistemskega gumba v zgornjem desnem delu okna, drugi način pa je z metodo *Close*. Stavek zapišemo npr. v telo odzivne metode dogodka *Click* nekega gumba na nemodalnem obrazcu, ki ga želimo zapreti.

```
private void bZapri_Click(object sender, EventArgs e)
{
     //zapiranje nemodalno odprtega obrazca
     this.Close();    //lahko zapišemo tudi samo stavek  Close();
}
```

Obrazec pa lahko zapremo tudi s klikom na gumb nekega drugega obrazca, ki sestavlja projekt. O tem bomo govorili v nadaljevanju, ko bomo povedali več o dostopnosti do gradnikov na drugih obrazcih.

V kolikor obrazca ne želimo zapreti in uničiti, ampak le začasno skriti, lahko uporabimo metodo *Hide*.

```
imeObrazca.Hide();
```

Če želimo kasneje obrazec zopet prikazati, ga ni potrebno na novo ustvariti (*stavek Form2 imeObrazca = new Form2();* ni potreben). Dovolj je le klic metode *Show* za ponoven prikaz obrazca. Ustvarjanje novega obrazca bi namreč pomenilo, da želimo ustvariti povsem nov objekt, katerega sestavni deli (polja, lastnosti, metode) nimajo nobeve zveze s skritim obrazcem.

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

Odpiranje pogovornih oken - modalnih obrazcev

Tudi odpiranje pogovornih oken smo že spoznali, ko smo delali z datotekami, pri izbiranju barv, pisave… Spoznali smo, da v tem primeru uporabimo metodo *ShowDialog*. Modalno odprti obrazci pri svojem zapiranju vračajo podatek o tem, kako smo jih zaprli. Zapremo jih lahko na dva načina: klasičen način je s pomočjo sistemskega gumba v zgornjem desnem delu okna, drugi način pa je ta, da je na obrazcu en ali več modalnih gumbov. Klik nanje povzroči, da se obrazec zapre. Ob tem metoda *ShowDialog* tudi vrne lastnost *DialogResult* tega gumba.

Modalni gumb je običajen gumb, ki ima nastavljeno lastnost *DialogResult*. Pri lastnosti *DialogResult* imamo na voljo naslednje opcije:

- *None* – privzeta vrednost gumba, okno se ob kliku na tak gumb ne zapre samodejno;
- *OK* - okno se zapre in vrne vrednost *DialogResult.OK*;
- *Cancel* - okno se zapre in vrne vrednost *DialogResult.Cancel*;
- *Abort* - okno se zapre in vrne vrednost *DialogResult.Abort*;
- *Retry* - okno se zapre in vrne vrednost *DialogResult.Retry*;
- *Ignore* - okno se zapre in vrne vrednost *DialogResult.Ignore*;
- *Yes* - okno se zapre in vrne vrednost *DialogResult.Yes*;
- *No* - okno se zapre in vrne vrednost *DialogResult.No*;

Še primer modalnega odpiranja obrazca na katerem sta npr. dva modalna gumba (gumba imata nastavljeno lastnost *DialogResult* na *OK* oziroma na *Cancel*):

```
Form2 novObrazec = new Form2();//ustvarjanje novega objekta – obrazca

/*Obrazec odpremo z metodo ShowDialog, ko pa ga uporabnik zapre, bo vrnil
neko vrednost, ki jo lahko testiramo kot pogoj v if stavku
if (novObrazec.ShowDialog() == DialogResult.OK) /*preverimo, kako je bil
                                                  obrazec zaprt*/
    MessageBox.Show("Gumb OK");//obrazec zaprt s klikom na modalni gumb OK
else MessageBox.Show("Gumb Prekliči");

/*else veja se izvede v primeru, da je bil obrazec zaprt s klikom na modalni
gumb Prekliči (lastnost DialogResult ima ta gumb nastavljeno na Cancel), ali
pa s klikom na sistemski gumb za zapiranje okna*/
```

Stavek *if (novObrazec.ShowDialog() == DialogResult.OK) …* moramo razumeti takole: objekt *novObrazec* izpeljan iz obrazca *Form2* smo odprli modalno z metodo *ShowDialog*. S tem obrazcem sedaj nekaj počnemo in ko ga bomo zaprli, se bo izvedlo preverjanje pogoja tega *if* stavka. Če bo metoda *ShowDialog* vrnila vrednost *DialogResult.OK* se izvede prva veja stavka *if*, sicer pa veja *else*. Veja *else* se bo izvedla tudi v primeru, da bo uporabnik zaprl okno s klikom na sistemski gumb za zapiranje okna. Klik na ta gumb namreč povzroči, da obrazec vrne vrednost *DialogResult.Cancel.*

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

Kadar ima modalni gumb "sprogramirano" metodo *Click* (ali kako drugo ustrezno metodo), se bo najprej izvedla ta metoda, šele nato se bo obrazec zaprl in vrnil temu gumbu nastavljeno lastnost *DialogResult*.

V primeru, da je na modalno odprtem obrazcu več modalnih gumbov, lahko preverjanje zapiranja obrazca izvedemo takole (v spodnjem primeru naj bi bili na modalno odprtem obrazcu modalni gumbi *OK*, *Retry* in *Cancel*):

```
Form2 novObrazec = new Form2();//ustvarjanje novega objekta – obrazca

/*obrazec odprimo z metodo ShowDialog; vrednost ki jo bo obrazec vrnil, ko ga
bo uporabnik zaprl se bo shranila v spremenljivko 'Gumb', ki je tipa
DialogResult*/
DialogResult Gumb = novObrazec.ShowDialog();

//preverimo, kateri gumb je uporabnik kliknil ko je zaprl obrazec
if (Gumb == DialogResult.OK)
    MessageBox.Show("Kliknjen je bil gumb OK");
else if (Gumb == DialogResult.Retry)
    MessageBox.Show("Kliknjen je bil gumb Ponovi");
else MessageBox.Show("Kliknjen je bil gumb Prekliči");
```

Napisani primer je le eden od možnih načinov preverjanja vrnjenih vrednosti modalnih gumbov. Lahko bi seveda preverjali le eno od vrnjenih modalnih vrednosti (npr *Dialogresult.OK*) in izvedli le eno vejitev, vse ostale pa bi npr. združili.

Metoda obrazca *Close* običajno ne zaključi aplikacije. Če je namreč odprtih več obrazcev, se aplikacija nadaljuje in konča tedaj, ko zapremo zadnji obrazec. Celotno aplikacijo pa lahko zapremo z metodo *Application.Exit()* ali pa *Environment.Exit(0)*. Pri uporabi te metode pa moramo vedeti, da se v tem primeru vsi odprti obrazci zaprejo. Pri tem zapiranju se metoda, prirejena dogodku *FormClosed* ne izvede. Zato so lahko v teh obrazcih (ki bi jih drugače shranili v metodi prirejeni *FormClosed*) neshranjeni podatki izgubljeni.

Vključevanje že obstoječih obrazcev v projekt

V poljuben projekt lahko dodamo tudi obrazec, ki že obstaja in je sestavni del nekega drugega projekta. V *Solution Explorerju* desno kliknem na trenuten projekt, izberemo opcijo *Add→Existing Item* in v pogovornem *Add Existing Item* poiščemo ustrezen obrazec. Izbiro potrdimo s klikom na gumb *Add* in vključitev je opravljena. V rešitev ki jo delamo, se ta datoteka (ali pa celoten projekt) tudi fizično skopira.

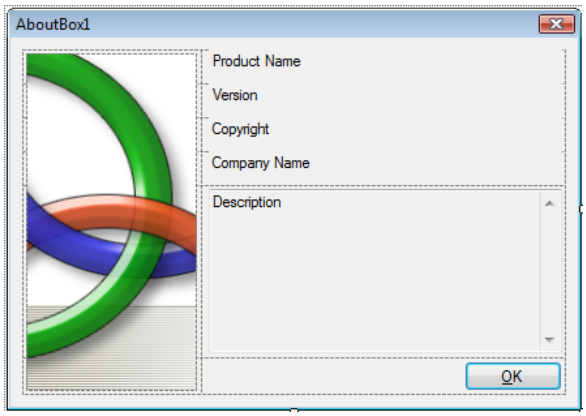
Odstranjevanje obrazca iz projekta

Podobno lahko iz kateregakoli projekta poljuben obrazec tudi izključimo. Postopek je podoben kot pri dodajanju. V oknu *Solution Explorer* obrazec najprej izberemo, kliknemo desni miškin gumb in nato izberemo opcijo *Delete*. Prikaže se pogovorno okno, v katerem lahko našo odločitev potrdimo ali pa prekličemo s klikom ustreznega gumba. Obstaja pa tudi možnost, da nek obrazec iz projekta le izključimo in ga tako fizično ne pobrišemo. To storimo tako, da namesto opcije *Delete* izberemo opcijo *Exclude From Project*.

Če hočemo nek že narejen obrazec dedovati, ga najprej v oknu *Solution Explorer* izberemo, kliknemo desni miškin gumb in v oknu ki prikaže izberemo opcijo *Inherit Form* (**Operacija je možna le polni verziji Visual C#, v Express Edition pa to ni mogoče!!!!!**)

Sporočilno okno AboutBox

Sporočilno okno *AboutBox* je okno, ki ga v projekt vključimo z namenom, da nam prikaže najpomembnejše podatke o projektu (ime, verzijo programa, opis, …). Okno dodamo v projekt tako, da v *Solution Explorerju* izberemo ustrezen projekt znotraj rešitve, kliknemo desni miškin gumb, izberemo opcijo *Add*, nato pa *New Item*. V oknu *Add New Item*, ki se odpre, izberemo *AboutBox*, po želji oknu tudi spremenimo ime. Dodajanje potrdimo s klikom na gumb *Add*. Sporočilno okno je tako dodano v naš projekt, Na obrazcu je nekaj oznak, okno s tekstom in slika.



Slika 12: Sprocilno okno *AboutBox*.

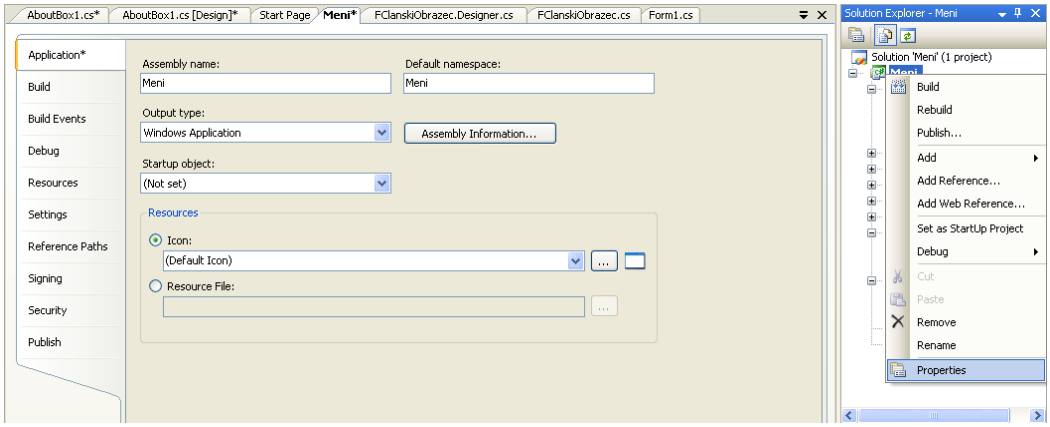
Lastnosti gradnikov *Label* in vsebino gradnika *TextBox* ne spreminjajmo, ker se bo njihova vsebina zapisala sama, oziroma jo bomo določili na drugem mestu, to je pri lastnostih projekta. Poglejmo kodo obrazca *AboutBox* (pogled *View Code*).

```
WindowsFormsApplication1.AboutBox1                    AboutBox1()
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Linq;
using System.Reflection;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    partial class AboutBox1 : Form
    {
        public AboutBox1()
        {
            InitializeComponent();
            this.Text = String.Format("About {0}", AssemblyTitle);
            this.labelProductName.Text = AssemblyProduct;
            this.labelVersion.Text = String.Format("Version {0}", AssemblyVersion);
            this.labelCopyright.Text = AssemblyCopyright;
            this.labelCompanyName.Text = AssemblyCompany;
            this.textBoxDescription.Text = AssemblyDescription;
        }

        Assembly Attribute Accessors
    }
}
```

Slika 13: Koda obrazca *AboutBox*.

Koda se je zgenerirala avtomatično: vsebuje kar nekaj klicev metod, ki so zapisane v delu *Assembly Attribute Accessors* – ta del kode je začasno skrit, seveda pa si skrito kodo lahko ogledamo s klikom na znak + na levem robu vrstice. Vseh stavkov verjetno ne bomo razumeli, a saj ni potrebno. Pomembno je le, da vemo, da koda v oknu *AboutBox* izpiše podatke o našem projektu – te podatke pa zapišemo v lastnostih projekta.

Lastnosti projekta nastavljamo v posebnem oknu, ki ga odpremo tako, da v *Solution Explorerju* izberemo naš projekt, nato pa z desnim miškinim gumbom odpremo *Pop Up Meni* in izberemo opcijo ***Properties***. Odpre se okno za nastavljanje lastnosti projekta.



Slika 14: Okno za nastavljanje lastnosti projekta.

V sekciji *Application* lahko nastavimo (spremenimo) glavne lastnosti projekta. S klikom na gumb *Assembly Information* se odpre okno za vnos dodatnih podatkov o tekočem projektu (ime, opis, verzija, …). Okno nam ponuja tudi možnost spreminjanja ikone, ki se bo kasneje prikazala v *Windows Explorerju* ob našem projektu.

V primeru, da naš projekt vsebuje več obrazcev, nam okno za nastavitev lastnosti projekta ponuja možnost, da izberemo, kateri obrazec se bo prikazal prvi. To storimo z izbiro v spustnem seznamu, ki se odpre v vrstici *Startup object*.

Okno vsebuje še cel kup lastnosti in razdelkov, ki pa jih zaenkrat ne bomo omenjali.

Za sporočilno okno *AboutBox* moramo sedaj le še napisati kodo za prikaz. To lahko storimo npr. z novo postavko v meniju, nanjo dvokliknemo in zapišemo ustrezno kodo. Okno odpremo modalno (metoda *ShowDialog*), zato da se bo zaprlo ob kliku na njegov gumb *OK*.

```
new AboutBox1().ShowDialog();
```

Lastnosti, ki smo jih nastavili v oknu *AboutBox* in spremembe, ki smo jih v njem naredili, bodo prikazane v oknu šele ob zagonu programa. Poženimo torej program in odprimo okno.

Dostop do polj, metod in gradnikov drugega obrazca

Dostop do polj in metod podrejenega obrazca

Pri izdelavi projekta z več obrazci se večkrat srečamo s problemom dostopa do polja (spremenljivke) ali pa do metode nekega drugega obrazca.

Spomnimo se, da do statičnih polj in do statičnih metod razreda (tudo obrazec je razred), dostopamo preko imena razreda (obrazca). Recimo da imamo obrazec *Form1* in iz njega odpiramo nov obrazec *Form2*. V razredu obrazca *Form2* naj obstaja statično polje *stevec* in statična metoda *Info*. Njuna definicija v obrazcu *Form2* je npr. takale:

```
public static int stevec = 0; //javna statična spemenljivka

public static string Info()
{
    return stevec + "\nRazred ima statično polje (stevec) in eno statično
                     metodo (Info)";
}
```

V obrazcu (razredu) *Form1* dostopamo do statičnih polj in statičnih metod razreda *Form2* tako, da najprej zapišemo ime obrazca (razreda), nato operator pika in končno še ime statičnega polja ali pa metode:

```
MessageBox.Show(Form2.Info() + "\nŠtevec objektov: " + Form2.stevec);
```

Do objektnih metod na drugih obrazcih, ali pa do javnih polj na drugih obrazcih pa dostopamo preko imena objekta, ki ga izpeljemo iz tega obrazca (razreda). Recimo da imamo obrazec *Form1* in iz njega odpiramo nov obrazec *Form2*. V razredu obrazca *Form2* naj obstaja javno polje *znesek* in javna metoda *Izpis*. Njuna definicija v obrazcu *Form2* je npr. takale:

```
public double znesek=0; //javna spemenljivka/polje tipa double
public void Izpis()//javna metoda Izpis
{
    MessageBox.Show("Trenutni znesek: " + znesek.ToString());
}
```

V obrazcu (razredu) *Form1* dostopamo do javnih polj in javnih metod razreda *Form2* tako, da iz razreda *Form2* najprej izpeljemo nov objekt, preko njega pa nato s pomočjo operatorja pika dostopamo do javnih polj in metod.:

```
Form2 novaForma = new Form2();//ime novega objekta je novaForma
double noviZnesek = novaForma.znesek; /*do polja znesek pridemo preko
                                        objekta novaForma
novaForma.Izpis();//tudi objektno metodo Izpis kličemo preko objekta
```

Dostop do gradnikov podrejenega obrazca

Ostane nam še opis načinov dostopa do gradnikov na drugem obrazcu. Ta je možen na dva načina:

- preko lastnosti *Controls* objekta, ki ga tvorimo iz nekega obrazca. Recimo da imamo obrazca *Form1* in *Form2*. Če želimo na obrazcu *Form1*, ki je trenutno odprt, dostopati do gradnikov na obratcu *Form2*, moramo v Form1 najprej ustvariti nov objekt tega tipa.

  ```
  Form2 novaForma = new Form2();//ime novega objekta je novaForma
  ```

  Do gradnikov objekta *novaForma* sedaj lahko dostopimo preko lastnosti *Controls* takole:

  ```
  /*dostop do gradnikov comboBox1 in textBox1, ki sta na obrazcu Form2,
  iz katerega smo izpeljali objekt novaForma*/
  novaForma.Controls["comboBox1"].Text = "Slovenija";
  novaForma.Controls["textBox1"].Text = "Kranj";
  ```

- preko imena objekta, ki smo ga izpeljali iz drugega obrazca – a v tem primeru morajo biti vsi gradniki, do katerih želimo imeti dostop deklarirani kot javni. Javno deklaracijo pa lahko določimo vsakemu gradniku na dva načina:

  - v datoteki *.Designer.cs* (v našem primeru je to datoteka *Form2.Designer.cs*) poiščemo ustrezno vrstico z deklaracijo tega gradnika (v tej datoteki je to nekje pri koncu, privzeta vrednost je *private*) in jo popravimo na public;

- najpogostejši in najlažji načina pa je ta, da vsakemu gradniku, do katerega želimo dostopati z nekega drugega obrazca, nastavimo lastnost *Modifiers* na *Public*.

```
/*dostop do gradnikov comboBox1 in textBox1, ki sta na obrazcu Form2
  Iz katerega izpeljemo objekt novaForma*/
Form2 novaForma = new Form2();//ime objekta je novaForma
/*dostop do gradnika comboBox1,ki pa mora biti javen*/
NovaForma.comboBox1.Text = "Slovenija";
/*dostop do gradnika textBox1,ki pa mora biti javen*/
NovaForma.textBox1.Text = "Kranj";
```

Kviz

Izdelajmo preprosti kviz. To bo večokenska aplikacija, ki bo uporabniku ponudila reševanje kviza. Vsebovala bo glavni obrazec, iz katerega bomo lahko odprli poljubno število modalnih podobrazcev. Število le-teh bo odvisno od števila vprašanj, ki jih bomo pripravili. Vprašanja bodo zapisana v tekstovni datoteki in bodo zato lahko iz različnih področij. Rezultate pa bomo shranjevali v svojo datoteko.

V ta namen bomo izdelali dva obrazca: glavni obrazec bo vseboval gradnik *MenuStrip* z dvema možnostima: *Začetek reševanja* (začetek reševanja kviza) in *Rezultati* (ogled dosedanjih rezultatov reševanja). V meniju naj bo še tekstovno polje, v katerega mora uporabnik pred začetkom reševanja vnesti svoje ime. Na glavnem obrazcu je tudi gradnik *Timer*, s pomočjo katerega spreminjamo barvo ozadja tekstovnega polja v meniju. S tem dosežemo učinek utripanja toliko časa, da uporabnik vnese svoje ime. Začetek reševanja kviza pred vnosom imena ni možen. Obrazcu priredimo tudi sliko za ozadje in lastnost *FormBorderStyle* nastavimo na *FixedToolWindow*.

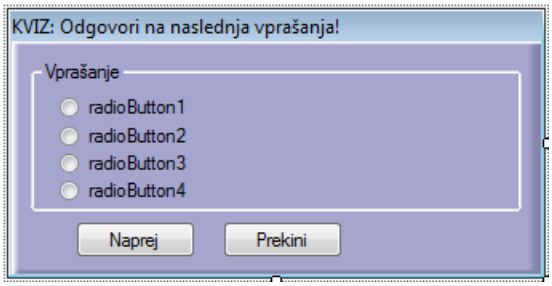
Slika 15: Glavni obrazec kviza.

Drugi obrazec se imenuje *FKviz* in vsebuje gradnik tipa *GroupBox* in v njem štiri radijske gumbe. Vsi štirje imajo lastnost *Modifiers* nastavljeno na *True*. Na obrazcu sta še gumba *Naprej* in

Prekliči. Obrazec *FKviz* bomo odpirali dinamično tolikokrat, kolikor je vprašanj v datoteki *Vprašanja.txt*. Ustvarimo jo kar v mapi *Bin→Debug* našega projekta: v vsaki vrstici te datoteke je najprej vprašanje, nato štirje možni odgovori, na koncu pa še številka, ki predstavlja pravilen odgovor. Posamezni podatki so med seboj ločeni s podpičjem. Tule sta za primer dve vrstici take datoteke:

```
Vsota notranjih kotov 4-kotnika je: ;180°;360°;440°;540°;2
V enakostraničnem trikotniku meri vsak kot: ; 30°;180°;60°;90°;3
```

Klik na gumb *Naprej* pomeni nadaljevanje kviza, klik na gumb *Prekliči* pa predčasen zaključek. Ko uporabnik odgovori na vsa vprašanja, naj se podatki o njegovem imenu in število pravilnih odgovorov zapišejo v tekstovno datoteko *Rezultati.txt*.



Slika 16: Obrazec *FKviz*.

Najprej napišimo odzivni metodi dogodkov *MouseEnter* in *MouseLeave* tekstovnega polja v glavnem meniju.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    /*ko se uporabik s kazalnikom miške zapelje čez tekstovno polje v meniju
      se zgodi dgodek MouseEnter*/
    private void toolStripTextBox1_MouseEnter(object sender, EventArgs e)
    {
        timer1.Enabled = false;
        toolStripTextBox1.BackColor = Color.GhostWhite;
        toolStripTextBox1.ForeColor = Color.Navy;
        if (toolStripTextBox1.Text == "Vnesi svoje ime...")
        {
            toolStripTextBox1.Clear();
        }
    }
    /*ko uporabik s kazalnikom miške zapusti tekstovno polje v meniju se
    zgodi dogodek MouseLeave*/
```

```
    private void toolStripTextBox1_MouseLeave(object sender, EventArgs e)
    {
        toolStripTextBox1.BackColor = Color.Gold;
        if ((toolStripTextBox1.Text.Trim()).Length == 0)
        {
            toolStripTextBox1.Text = "Vnesi svoje ime...";
            timer1.Enabled = true;
        }
        else timer1.Enabled = false;
    }
}
```

Učinek utripanja tekstovnega polja za vnos imena bomo dosegli s pomočjo gradnika *Timer*.

```
/*pomožna logična spremenljivka, s pomočjo katere spreminjamo barvo teksta in
  s tem dosežemo utripanje!*/
bool utripaj = true;
private void timer1_Tick(object sender, EventArgs e)
{
    //če ime še ni vnešeno
    if (toolStripTextBox1.Text == "Vnesi svoje ime...")
    {
        if (utripaj)
            toolStripTextBox1.ForeColor = Color.Navy;
        else
            toolStripTextBox1.ForeColor = Color.Red;
        utripaj = !utripaj;
    }
}
```

Pred pisanjem odzivne metode, za začetek reševanja kviza, pripravimo kodo obrazca *FKviz*. Konstruktor obrazca naj sprejme štiri parametre: vprašanje in možne odgovore. Vprašanje bomo priredili lastnosti *Text* gradnika tipa *GroupBox*, odgovore pa radijskim *gumbom*.

```
public partial class FKviz : Form
{
    /*Konstruktor obrazca FKviz prejme za parametre vprašanje in štiri možne
      Odgovore*/
    public FKviz(string vprasanje, string odgovor1, string odgovor2, string
odgovor3, string odgovor4)
    {
        InitializeComponent();

        /*Vprašanje priredimo lastnosti Text gradnika tipa GroupBox, odgovore
          pa radijskim gumbom*/
        groupBox1.Text = vprasanje;
        radioButton1.Text = odgovor1;
        radioButton2.Text = odgovor2;
        radioButton3.Text = odgovor3;
```

```
        radioButton4.Text = odgovor4;
    }
    //Ko se obrazec prikaže, prvi radijski gumb ne bo izbran
    private void FKviz_Shown(object sender, EventArgs e)
    {
        radioButton1.Checked = false;
    }

    /*Ob kliku na gumb Naprej, preverimo, če je uporabnik izbral enega od
      ponujenih odgovorov. Obrazec se v tem primeru zapre, sicer pa ne!*/
    private void button1_Click(object sender, EventArgs e)
    {
        if (radioButton1.Checked || radioButton2.Checked ||
            radioButton3.Checked || radioButton4.Checked)
            DialogResult = DialogResult.OK; //Obrazec se zapre
        else
        {
            DialogResult = DialogResult.None; //Obrazec ostane odprt
            MessageBox.Show("Nisi izbral odgovora!");
        }
    }
}
```

V odzivni metodi dogodka *Shown* smo poskrbeli, da po prikazu obrazca ne bo izbran noben radijski gumb. Privzeto bi bil sicer izbran prvi gumb. Uporabnik obrazca ne more zapreti toliko časa, da je izbral enega od odgovorov. Za to smo poskrbeli v odzivni metodi gumba za zapiranje tega obrazca.

Začetek reševanja kviza bomo obdelali v odzivni metodi dogodka *Click*. Ta se izvede ob uporabnikovem kliku na gumb za začetek reševanja kviza. V metodi najprej preverimo, če je uporabnik vnesel svoje ime. Sledi branje datoteke z vprašanji. V zanki beremo vsako vrstico posebej in s pomočjo metode *Split* ločimo vprašanja in odgovore. Nato ustvarimo nov objekt razreda *FKviz*, ki mu s pomočjo konstruktorja pošljemo vprašanje in štiri možne odgovore. Sledi preverjanje pravilnosti odgovora in če je kviz končan, še zapis rezultatov v tekstovno datoteko.

```
private void kVIZToolStripMenuItem_Click(object sender, EventArgs e)
{
    try
    {
        if (toolStripTextBox1.Text == "Vnesi svoje ime...")
            MessageBox.Show("Pred začetkom reševanja moraš vnesti svoje
ime!", "POZOR!", MessageBoxButtons.OK, MessageBoxIcon.Information);
        else
        {
            /*datoteko Vprašanja.txt z vprašanji odpremo za branje:
              datoteka se nahaja v mapi Bin->Debug našega projekta*/
            StreamReader beri = File.OpenText("Vprašanja.txt");
            string vrstica = beri.ReadLine();
```

```
            int zaporedna = 1;//zaporedna številka obrazca/vprašanja
            int pravilnihOdgovorov = 0;
            while (vrstica != null) //dokler so podatki v datoteki
            {
                /*V vsaki vrstici datoteke je najprej vprašanje, nato štirje
                  možni odgovori, na koncu pa številka pravilnega odgovora.
                  Posamezne postavke so ločene s podpičjem. Razločimo jih z
                  metodo Split*/
                string[] besedilo = vrstica.Split(';');
                /*v prvi celici tabele bo vprašanje, v naslednjih štirih
                  možni odgovori, v zadnji celici pa številka pravilnega
                  odgovora*/
                /*Naredimo nov objekt razreda (obrazca) FKviz in mu kot
                  parametre pošljemo vprašanje in možne odgovore*/
                FKviz novi = new FKviz(zaporedna + ". " + besedilo[0],
besedilo[1], besedilo[2], besedilo[3], besedilo[4]);
                //objekt novi odpremo kot dialog
                if (novi.ShowDialog() == DialogResult.Abort)
                {
                    /*če uporabnik klikne gumb Prekini, se reševanje
                    kviza zaključi*/
                    MessageBox.Show("Reševanje kviza bo prekinjeno!");
                    beri.Close();
                    zaporedna = 0;
                    break;
                }
                /*Če je bil obrazec zaprt s klikom na gumb Naprej, se kviz
                  nadaljuje: Preverimo, če je odgovor pravilen. Številka
                  pravilnega odgovora se nahaja v celici besedilo[5]*/
                switch (Convert.ToInt32(besedilo[5]))
                {
                    /*vsi radijski gumbi na obrazcu FKviz morajo imeti
                      lastnost Modifiers nastavljeno na True. Do njihovih
                      lastnosti lahko dostopimo preko objekta novi*/
                    case 1: if (novi.radioButton1.Checked)
                              pravilnihOdgovorov++;
                              break;
                    case 2: if (novi.radioButton2.Checked)
                              pravilnihOdgovorov++;
                              break;
                    case 3: if (novi.radioButton3.Checked)
                              pravilnihOdgovorov++;
                              break;
                    case 4: if (novi.radioButton4.Checked)
                              pravilnihOdgovorov++;
                              break;
                }
                zaporedna++; //zaporedna številka vprašanja se poveča
                vrstica = beri.ReadLine(); //beremo naslednjo vrstico
            }
```

```
            beri.Close();//zapremo datoteko z vprašanji
            if (zaporedna > 0)/*preverimo, če je bilo kaj vprašanj, oz.
                              reševanje ni bilo prekinjeno*/
            {
                MessageBox.Show("Število pravilnih odgovorov: " +
pravilnihOdgovorov);
                //Rezultat zapišemo še v datoteko Rezultati.txt. Datoteke se
                //nahaja v mapi Bin->Debug našega projekta
                File.AppendAllText("Rezultati.txt", toolStripTextBox1.Text +
" - Pravilni odgovori: " + pravilnihOdgovorov + ", napačni odgovori: " +
(zaporedna - pravilnihOdgovorov) + "\n");
            }
        }
    }
    catch
    {
       MessageBox.Show("Napaka pri delu z datoteko!");
    }
}

/*metoda za branje datoteke dosedanjih rezultatov in prikaz le-teh v
  sporočilnem oknu*/
private void rezultatiToolStripMenuItem_Click(object sender,EventArgs e)
{
    /*preberemo celotno vsebino datoteke Rezultati.txt. Datoteka se nahaja v
      mapi Bin->Debug našega projekta*/
    string vsebina = File.ReadAllText("Rezultati.txt");
    MessageBox.Show(vsebina);
}
```

Dodali smo še odzivno metodo za branje in prikaz vsebine datoteke z dosedanjimi odgovori. Vsebino te datoteke smo prikazali kar v sporočilnem oknu.

Dostop do polj, metod in gradnikov nadrejenega obrazca

Če hočemo v podrejenih obrazcih priti do polj, metod in gradnikov nadrejenega obrazca, je potreben poseg v projektno datoteko. Običajno je njeno ime *Program.cs*. Vsebino projektne datoteke spremenimo tako, da naredimo objekt, ki ga izpeljemo iz glavnega obrazca, *statičen*. Že iz osnov OOP namreč vemo, da do statični polj dostopamo neposredno preko imena razreda.

Recimo, da se glavni obrazec projekta imenuje *FGlavni*. Spremenimo ga takole:

```
/*napoved statičnega objekta 'Glavni' izpeljanega iz razreda FGlavni ->
 static public je zato, da bomo do njegovih gradnikov in polj imeli dostop
 tudi iz podrejenih obrazcev*/
public static FGlavni Glavni;

[STAThread]
```

```
static void Main()
{
     Application.EnableVisualStyles();
     Application.SetCompatibleTextRenderingDefault(false);
     //objekt Glavni še dejansko ustvarimo
      Glavni = new FGlavni();
     Application.Run(Glavni);//in ga odpremo kot začetni obrazec v projektu
}
```

Iz glavnega obrazca smo odprli nov obrazec (objekt z imenom *Glavni*), izpeljan iz razreda *FPodrejeni*. V kateremkoli drugem obrazcu našega projekta, lahko sedaj dostopimo do polj, metod in gradnikov glavnega obrazca tako, da zapišemo ime razreda, iz katerega je izpeljan glavni obrazec.

```
Program.Glavni.ime;
```

Tu je *ime* ime polja, metode ali pa gradnika na nadrejenem obrazcu.

Garbage Collector

Uporaba Garbage Collectorja in upravljanje s pomnilnikom

Nov primerek (*instanco*) nekega razreda, oziroma nov objekt, ustvarimo s pomočjo rezervirane besede *new*. Poznamo jo že iz osnov OOP. Vemo že, da je ustvarjanje objekta dvofazni proces. Z operatorjem *new* najprej alociramo (zasedemo) prosti pomnilnik na kopici, zasedeni pomnilnik pa je nato potrebno še pretvoriti v objekt in ga inicializirati. Pri inicializaciji si lahko pomagamo s konstruktorjem in ko je nov objekt ustvarjen, lahko do njegovih članov (polj, metod, ..) dostopamo z uporabo operatorja pika. "Spremenljivke", ki jih ustvarimo z operatorjem new so referenčne spremenljivke. Nahajajo se na kopici, pravimo jim tudi *objekti*.

```
//Ustvarjanje novega objekta - TextBox je referenčni tip(ustvarjen na kopici)
TextBox sporocilo = new TextBox();
//do lastnosti objekta sporocilo pridemo  s pomočjo operatorja pika
sporocilo.Text = "Tekstovno sporočilo!";
```

Iz tako ustvarjenega objekta lahko v nadaljevanju tvorimo poljubno število novih referenc:

```
TextBox spor1 = sporocilo;/*s sklicevanjem na že ustvarjeni objekt lahko
                           naredimo tudi novo referenčno spremenljivko*/
```

Vse te reference pa je potrebno slediti, oz. imeti nad njimi popoln nadzor. Če spremenljivka *sporocilo* npr. izgine oz. ni več dostopna(zaključek bloka, zaključek metode, …), lahko ostale spremenljivke (v našem primeru npr. spremnljivka *spor1*) še vedno obstajajo. Življenjska doba objekta torej ni vezana na določeno referenčno spremenljivko. Objekt se lahko uniči šele tedaj, ko izginejo vse njegove reference.

Destruktorji in Garbage Collector

Tako kot je dvofazni proces ustvarjanje novega objekta, je dvofazni proces tudi njegovo uničenje – je njegova zrcalna slika. Prvi del "čiščenja" opravimo s pisanjem *destruktorja*. Drugi del faze pa je v tem, da je potrebno ob uničenju objekta sprostiti (alocirati) del pomnilnika, ki ga je objekt zasedal in ga vrniti kopici v nadaljnjo prosto uporabo. Nad to drugo fazo, tako kot pri ustvarjanju objekta, pa nimamo neposrednega vpliva. Proces uničenja objekta in vračanje pomnilnika kopici je poznano pod imenom *garbage collection*.

V okolju Visual C# objektov nikoli ne moremo uničiti sami. Za to opravilo ne obstaja nikakršna sintaksa, tako kot je npr. v C++ temu namenjena metoda *delete*. Kar nekaj pomembnih razlogov botruje temu, da nam C# ne omogoča eksplicitno uničenje objektov. V primeru, da bi bila odgovornost nad uničenjem objektov prepuščena nam, bi slej ko prej prišlo do ene od naslednjih situacij:

- pozabili bi uničiti nek objekt. To bi pomenilo, da se objektov destruktor ni izvedel, čiščenje pomnilnika se ni izvedlo, s tem pa pomnilnik na kopici, ki ga objekt zaseda, ne bi bil sproščen. Na ta način bi kmalu ostali brez pomnilnika;
- poskusili bi uničiti aktiven objekt, kar pa bi bila katastrofa za vse objekte z referenco na ta uničeni objekt;
- isti objekt bi lahko poskušali uničiti večkrat, kar bi bilo prav tako lahko katastrofalno, odvisno od destruktorja.

Zgornji problemi so torej nesprejemljivi. Za uničevanje objektov je zato odgovoren proces *garbage collector*. Ta proces nam zagotavlja, da bo vsak objekt zagotovo uničen, obenem pa se bo izvedel njegov destruktor.

Če napišemo destruktor, se bo ta zagotovo izvedel, ne vemo pa kdaj, saj o uničenju objektov odloča *garbage collector*. Ko se zaključi nek program, bodo zagotovo uničeni tudi vsi v programu ustvarjeni objekti. Obenem proces zagotavlja, da bo vsak objekt uničen natanko enkrat. Uničen bo samo takrat, ko postane nedostopen – to pa pomeni tedaj, ko ne obstaja več nobena referenca na ta objekt.

Ostane pa še vprašanje, kdaj pa se čiščenje (*garbage collection*) dejansko izvede. Prav gotovo je to tedaj, ko objekt ni več potreben. To pa se ne zgodi vedno neposredno za tem, ko objekta ne potrebujemo več, npr. neposredno po zaključku nekega bloka. Zažene se tedaj, ko sistem zazna, da mu začne primanjkovati pomnilnika, oz. da je prezaposlen. Tedaj sprosti pomnilnik vseh

tistih objektov, ki niso več v funkciji. Obstaja tudi način, da proces *garbage collector* zaženemo sami. To lahko naredimo s stavkom

```
System.GC.Collect();
```

Tak ekspliciten zagon procesa *garbage collection* ni priporočljiv. Proces se sicer res zažene, a ker se izvaja asinhrono, po njenem zaključku še vedno ne vemo, ali so bili vsi objekti res uničeni. Uničevanje objektov je zato najpametneje prepustiti kar procesu *Garbage Collection*.

Kako deluje Garbage Collector

Garbage Collector teče v svoji lastni niti in se lahko izvede samo v določenem času (tipično npr. ob zaključku neke metode). Ostale niti, ki tečejo istočasno v programu, se tedaj začasno zaustavijo. G*arbage collector* bo morda premikal posamezne objekte in ažuriral njihove reference, to pa ni možno tedaj, ko so objekti v uporabi.

Upravljanje s pomnilnikom

S pisanjem destruktorjev postane koda bolj kompleksna, bolj zahteven postane tudi proces *garbage collection*, program pa počasnejši. Pisanja *destruktorjev* se zaradi tega izogibamo, včasih pa njihovo pisanje še posebej ni priporočljivo.

Nekateri viri pa so tako pomembni, da njihovo sproščanje ni pametno prepustiti *garbage collectorju* – potrebno jih je sprostiti čim prej je to mogoče. Metodi, ki poskrbi za sproščanje nekega vira, pravimo metoda za odstranjevanje (*disposal method*). V primeru, da nek razred vsebuje *dispose* metodo, lahko le-to pokličemo eksplicitno, s tem pa imamo nadzor nad sproščanjem ustreznega vira. Namesto pisanja *destruktorja* zato v takih primerih raje uporabimo t.i. *using* stavek.

Using stavek

Using stavek predstavlja univerzalni mehanizem za kontrolo življenjske dobe objektov (virov). Katerikoli objekt, ki ga ustvarimo v glavi *using* bloka bo avtomatično uničen, ko se ta blok konča. (*POZOR: using stavka ne smemo zamenjati z ukazom using, s katerim v projekt dodamo nek imenski prostor.*)

Splošna sintaksa *using* stavka:

```
using (deklaracija objekta = inicializacija) stavki;
```

Tipičen primer uporabe *using* stavka je pri delu z datotekami. Objekt za branje ali pisanje definiramo v glavi using stavka, v telesu pa ta objekt uporabljamo:

```
//v glavi using stavka napovemo in ustvarimo nov objekt
```

```
using (StreamReader branje=new StreamReader(imeDatoteke))
{
    string vrstica, vsebina = "";
    //beremo dokler ne zmanjka vrstic
    while ((vrstica = branje.ReadLine()) != null)
       vsebina += vrstica;
} /*konec bloka using stavka - avtomatski klic metode, ki poskrbi za
    sproščanje pomnilnika, ki ga zaseda objekt branje*/
```

Če bi hoteli zgornji *using* stavek nadomestiti z enakovredno kodo, brez uporabe *using* stavka, bi jo morali zapisati takole:

```
try
{
    //ustvarimo nov objekt za branje odatkov iz tekstovne datoteke
    StreamReader branje = new StreamReader("Datoteka.txt");
    try
    {
        string vrstica, vsebina = "";
        while ((vrstica = branje.ReadLine()) != null)
            vsebina += vrstica;
    }
    finally  //brezpogojni varovalni blok
    {
        branje.Close();
    }
}
catch
{
    MessageBox.Show("Napaka!");
}
```

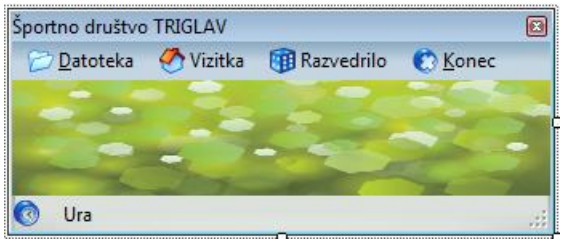
Tak način je sicer legalen in v zgornjem primeru tudi ustrezen. Problem pa nastane, kadar je potrebno odstranjevanje (sproščanje) več kot enega vira. V takem primeru bi bilo potrebno gnezdenje varovalnih blokov. Poleg tega je v zgornjem primeru referenca na objekt *branje* ostala tudi potem, ko se varovalni blok že zaključi. Za reševanje takih problemov je zato v C# boljša uporaba *using* stavka.

Kadar je projekt sestavljen iz več obrazcev in v teh obrazcih uporabljamo objekte, ki jih izpeljemo iz naših lastnih razredov, je priporočljivo, da razred napišemo v svoji knjižnici ali pa v svoji *.cs* datoteki, ločeno od obrazca. Tak način je bil razložen v prejšnjih poglavjih. To naredimo tako, da Izberimo opcijo *Project -> Add Class* in nato v pogovornem oknu (možnost *Class* je v tem primeru že izbrana), ter določimo ime datoteke. Ime lahko pustimo tudi privzeto, C*lass1.cs*. Za vstavljanje nove datoteke z razredom pa obstaja še ena pot: v *SolutionExplorerju* desno kliknemo na tekoči projekt in izberimo opcijo *Add→ NewItem*. Odpre se enako pogovorno okno kot pri prejšnjem načinu: izberimo možnost *Class* in določimo ime novega razreda.

Evidenca članov športnega društva

Pripravimo projekt, s pomočjo katerega bomo vodili evidenco članov športnega društva. Dodajali bomo lahko nove člane, urejali njihove podatke, lahko jih bomo brisali iz evidence, možen pa bo tudi tabelarični izpis vseh članov in še kaj. Program bo uporabljal eden ob obstoječih članov v svojem prostem času, zato bomo poskrbeli tudi za njegovo razvedrilo. V program bomo vključili v enem od prejšnjih poglavij napisano igro *Tri v vrsto*.

Projekt bodo sestavljali trije novi obrazci (*FGlavni*, *FPregled* in *FClanskiObrazec*) in dva že obstoječa obrazca (*FTriVVrsto* in *AboutBox*). Na glavni obrazec dodajmo sliko za ozadje, nato pa vrstico z menijem, gradnik *StatusStrip* in gradnik *Timer* za prikaz ure. V mapo *Resources* dodajmo še nekaj sličic na enak način, kot smo storili z glasbenimi datotekami v vaji *Predvajalnik glasbe in videa*. Sličice uporabimo za grafično obogatitev vrstice z menijem.



Slika 17: Glavni obrazec projekta *Evidenca članov športnega društva*.

Meni na glavnem obrazcu je sestavljen iz naslednjih možnosti:

- Datoteka
  - Dodaj
  - Pregled in ažuriranje
- Vizitka
- Razvedrilo
  - Tri v vrsto
- Konec

Za potrebe urejanja članstva, prenosa podatkov iz in v datoteko pripravimo razred *Clan.cs*. V oknu *Solution Explorer* desno kliknimo na projektno datoteko, izberemo *Add→New Item* in nato v oknu *Add New Item* izberemo *Class*. Poimenujmo ga *Clan.cs* in kliknimo gumb *Add*. Vsebino datoteke zapišimo takole:

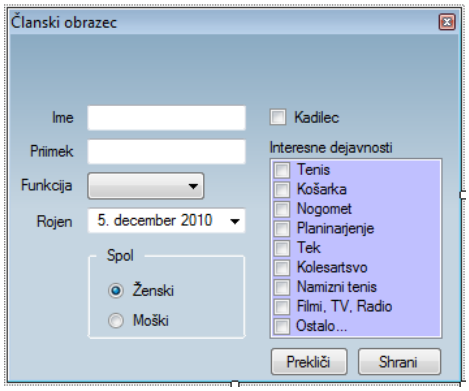
```
class Clan
{
    //javna polja razreda Clan
    public string ime, priimek;
```

```
    public string funkcija;
    public DateTime rojstvo;
    public char spol;
    public bool kadilec;
    public ArrayList dejavnosti; //netipizirana zbirka za zapis dejavnosti
    //objektna metoda, ki vrne niz z vsemi podatki o članu
    public string Podatki()//metoda vrne niz z vsemi podatki o članu
    {
        string clanskiPodatki = "Ime: " + ime + ", Priimek: " + priimek + ",
                                Funkcija: " + funkcija + ", Spol: ";
        if (spol=='Z')
            clanskiPodatki += "Ženski";
        else
            clanskiPodatki += "Moški";
        clanskiPodatki += ", Kadilec: ";
        if (kadilec)
            clanskiPodatki = clanskiPodatki + "Da\n";
        else clanskiPodatki = clanskiPodatki + "Ne\n";
        return clanskiPodatki;
    }
}
```

Dodali smo še objektno metodo *Podatki*, ki vrne niz z vsemi podatki o določenem članu.

Ker je razred *Clan* vključen v naš projekt, ga lahko sedaj uporabljamo kjerkoli znotraj projekta. Razred bi seveda lahko zapisali tudi v svojo knjižnico, tako kot smo to naredili v enem od prejšnjih projektov.

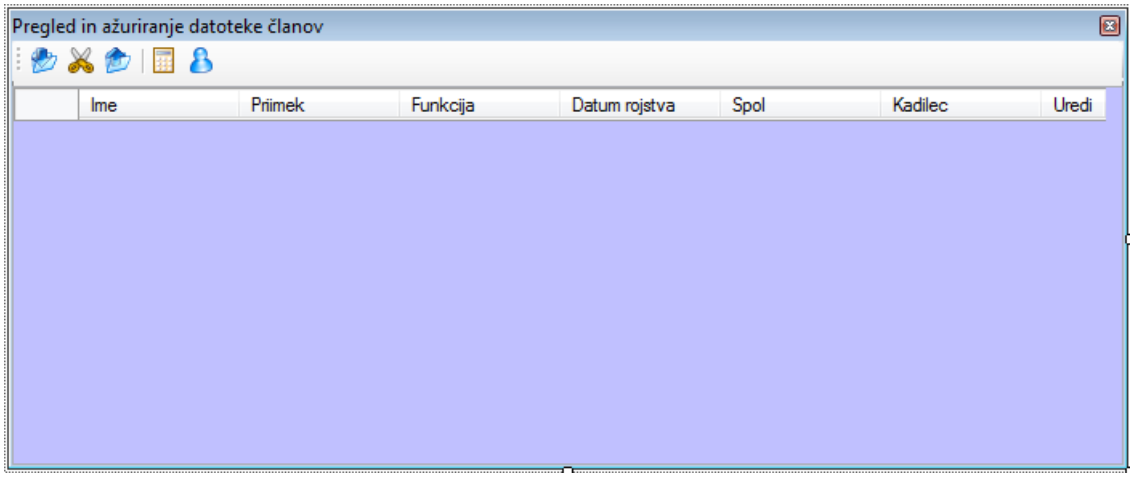
Za dodajanje oz. ažuriranje podatkov, ter za pregled članstva pripravimo dva nova obrazca: *FClanskiObrazec* in *Fpregled*. Na članski obrazec postavimo dva gradnika tipa *TextBox*, nato pa še *ComboBox*. Ta naj ima lastnost *DropDownStyle* nastavljeno na *DropDownList*, v *EditItems* pa vnesemo vrstice *Član*, Predsednik, *Podpredsednik*, *Tajnik*, *Blagajnik* in *Trener*. Dodajmo še gradnike tipov *DatetTimePicker*, *GroupBox* z dvema radijskima gumboma, *CheckBox*, *CheckedListBox* in dva gumba. V gradnik *CheckBox* s pomočjo lastnosti *Items* dodajmo nekaj najpopularnejših športov.



Slika 18: Članski obrazec *FClanskiObrazec*.

Na obrazec postavimo tudi gradnik *ErrorProvider*, preko katerega bomo uporabniku pri zapiranju tega obrazca sporočali, ali so vnosi pravilni. Gumb z napisom *Prekliči* naj ima lastnost *DialogResult* nastavljeno na *Cancel*, gumb *Shrani* pa lastnost *DialogResult* enako *OK*.

Na obrazec *FPregled*, ki je namenjen pregledu in urejanju obstoječih članov, najprej postavimo vrstico z menijem. V ta meni dodajmo gumbe, ki jim bomo priredili odzivne metode za ažuriranje vsebine gradnika *DataGridView*, brisanje izbrane vrstice, shranjevanje v datoteko. Zadnja dva gumba bosta namenjena obdelavi gradnika *DataGridView*: ob kliku na prvi gumb želimo ugotoviti skupno število kadilcev, ob kliku na drug gumb pa skupno število mladoletnih članov. Rezultat bomo v obeh primerih zapisali v sporočilnem oknu. Preostali del obrazca zapolnjuje gradnik *DataGridView*, v katerem naredimo stolpce *Ime*, *Priimek*, *Funkcija*, *Rojstvo* (napis na stolpcu je *Datum rojstva*), *Spol*, *Kadilec*, *Dejavnosti* in *Uredi*. Vsi stolpci so tipa *TextBox*, stolpec *Uredi* pa je tipa *Button*. Stolpec *Dejavnosti* ima lastnost *Visible* nastavljeno na *False* in zato ni viden. Ob kliku na gumb *Uredi* želimo odpreti obrazec za urejanje podatkov izbranega člana, podatke na njem urediti in shraniti nazaj na isto mesto. V gradniku *DataGridView* onemogočimo neposredno dodajanje, urejanje in brisanje vrstic s podatki.



Slika 19: Obrazec za pregled in urejanje članov športnega društva.

Pred pisanjem kode posameznih obrazcev v projekt dodajmo še obrazec, ki smo ga ustvarili v enem od prejšnjih projektov. V *Solution Explorerju* desno kliknimo na projektno datoteko, izberimo *Add→Existing Item*. Poiščimo projekt *TriVVrsto* in v njem izberemo obrazec *TriVVrsto.cs.* S klikom na gumb *Add* ga dodajmo v projekt. Končno v projekt dodajmo še okno *AboutBox* (desni klik na projektno datoteko, *Add→New Item→AboutBox*), ki mu po navodilih iz prejšnjega poglavja nastavimo ustrezne atribute.

Za dodajanje novega člana v seznam bomo napisali odzivno metodo postavke *Dodaj* glavnega menija. Obrazec s podatki o posameznem članu smo že pripravili. Poskrbimo za to, da bo uporabnik zagotovo vnesel svoje ime in priimek, ter funkcijo, ki jo ima v klubu. V ta namen napišimo dve lastni metodi, v katerih bomo uporabili metodo *SetError* gradnika *ErrorProvider*. Koda članskega obrazca bo takale:

```
public FClanskiObrazec()
{
    InitializeComponent();
    /*POZOR: gradnik cBFunkcija ima lastnost DropDownStyle nastavljeno
      na DropDownList (uporabnik lahko vrednost le izbira, ne more pa
      vnesti druge vrednosti!!! S pomočjo lastnosti EditItems smo v ta
      gradnik vnesli postavke Član, Predsednik, Podpredsednik, Tajnik,
      Blagajnik in Trener*/
}
//klik na gumb Shrani za zapiranje obrazca
private void Shrani_Click(object sender, EventArgs e)
{
    /*obrazec ostane odprt če uporabnik ni vnesel imena in priimka in
      izbral funkcije*/
    if (KontrolaImenaInPriimka() && KontrolaFuncije())
        DialogResult = DialogResult.OK;
    else
    {
        DialogResult = DialogResult.None;
    }
}
/*lastna metoda za kontrolo vnosa imena: metoda vrne False, če vnos ni
  popoln*/
private bool KontrolaImenaInPriimka()
{
    //če uporabnik ni vnesel imena in priimka, ga obvestimo o napaki
    if (tBIme.Text == "" || tBPriimek.Text == "")
    {
        if (tBIme.Text == "")
            errorProvider1.SetError(tBIme, "Nisi vnesel imena!");
        if (tBPriimek.Text == "")
            errorProvider1.SetError(tBPriimek, "Nisi vnesel Priimka!");
        return false;
    }
    else //sicer pa obvestilo o napaki odstranimo
    {
        errorProvider1.SetError(tBIme, "");
        errorProvider1.SetError(tBPriimek, "");
        return true;
    }
}
/*lastna metoda za kontrolo izbire funkcije v klubu: metoda vrne False, če
  ni izbrana nobena od ponujenih možnosti*/
private bool KontrolaFuncije()
{
    //če uporabnik ni izbral funkcije, ga obvestimo o napaki
    if (cBFunkcija.Text == ""  )
    {
        errorProvider1.SetError(cBFunkcija, "Nisi izbral funkcije!");
        return false;
```

```
    }
    else //sicer pa obvestilo o napaki odstranimo
    {
        errorProvider1.SetError(cBFunkcija, "");
        return true;
    }
}
```

Preden se lotimo pisanja kode glavnega obrazca poskrbimo še za odzivne metode obrazca *FPregled*. Odzivno metodo dogodka *Load* uporabimo zato, da iz datoteke *SDTriglav.txt* prenesemo podatke o članstvu v *DataGridView*. Objekt za branje ustvarimo v *using* stavku. Napišimo tudi odzivno metodo dogodka *CellClick* za ažuriranje podatkov izbrane vrstice in odzivno metodo za brisanje izbrane vrstice gradnika *DataGridView*. Pred zaprtjem obrazca še poskrbimo, da se podatki o članstvu shranijo nazaj v isto datoteko.

```
public partial class FPregled : Form
{
    public FPregled()
    {
        InitializeComponent();
        /*dataGridView1 ima lastnost SelectionMode nastavljeno na
          FullRowSelect*/
    }
    /*preden se obrazec odpre iz datoteke SDTriglav preberemo podatke
      o članstvu in jih prenesemo v gradnik DataGridView.*/
    private void FAzuriraj_Load(object sender, EventArgs e)
    {
        dataGridView1.Rows.Clear();
        string imeDatoteke = "SDTriglav.txt"; //določimo ime datoteke
        //če datoteka že obstaja, njeno vsebino prenesemo v dataGridView1
        if (File.Exists(imeDatoteke))
        {
            /*Using stavek predstavlja univerzalni mehanizem za kontrolo
              življenjske dobe objekta beri, za branje datoteke. Objekt
              ustvarimo v glavi using bloka in bo avtomatično uničen, ko se
              ta blok konča.*/
            using (StreamReader beri = File.OpenText(imeDatoteke))
            {
                // skušamo prebrati prvo vrstico
                string vrstica = beri.ReadLine();
                while (vrstica != null)  // dokler je branje vrstic uspešno
                {
                    /*nov objekt razreda Clan: razred je zapisan v datoteki
                      Clan.cs*/
                    Clan clan = new Clan();
                    /*podatke v vrstici, ki smo jo prebrali iz datoteke, z
                      metodo Split razporedimo v tabelo nizov*/
                    string[] tab = vrstica.Split('|');
                    clan.ime = tab[0];
```

```
                    clan.priimek = tab[1];
                    clan.funkcija = tab[2];
                    clan.rojstvo = Convert.ToDateTime(tab[3]);
                    clan.spol = Convert.ToChar(tab[4]);
                    clan.kadilec = Convert.ToBoolean(tab[5]);
                    clan.dejavnosti = new ArrayList();
                    for (int j = 0; j < tab.Length - 6; j++)
                        clan.dejavnosti.Add(tab[j + 6]);
                    //polja objekta clan zapišemo v DataGridView
                    dataGridView1.Rows.Add(clan.ime, clan.priimek,
clan.funkcija, clan.rojstvo, clan.spol, clan.kadilec, clan.dejavnosti);
                    // skušamo prebrati naslednjo vrstico
                    vrstica = beri.ReadLine();
                }
            }
        }
    }
    //odzivna metoda za ažuriranje odatkov izbrane vrstice
    private void dataGridView1_CellClick(object sender,
DataGridViewCellEventArgs e)
    {
        if (e.ColumnIndex == 7)//Preverim, če je kliknjen gumb Uredi
        {
            //številka vrstice gradnika dataGridView1
            int vrstica = dataGridView1.CurrentRow.Index;
            //vrednosti izbrane vrstice moramo prenesti na članski obrazec
            FClanskiObrazec FClanskiObrazec = new FClanskiObrazec();
            FClanskiObrazec.tBIme.Text =
                        dataGridView1.CurrentRow.Cells[0].Value.ToString();
            FClanskiObrazec.tBPriimek.Text =
                        dataGridView1.CurrentRow.Cells[1].Value.ToString();
            FClanskiObrazec.cBFunkcija.Text =
                        dataGridView1.CurrentRow.Cells[2].Value.ToString();
            FClanskiObrazec.dTPRojen.Value =
                 Convert.ToDateTime(dataGridView1.CurrentRow.Cells[3].Value);
            if (dataGridView1.CurrentRow.Cells[4].Value.ToString() == "Z")
                FClanskiObrazec.radioButton1.Checked = true;
            else
                FClanskiObrazec.radioButton2.Checked = true;

            FClanskiObrazec.cBKadilec.Checked =
Convert.ToBoolean(dataGridView1.CurrentRow.Cells[5].Value);

            Clan clan = new Clan();
            /*celico z indeksom 6, v kateri je zbirka dejavnosti, prenesemo
              v objekt clan*/
            clan.dejavnosti =
(ArrayList)dataGridView1.CurrentRow.Cells[6].Value;

            for (int n = 0; n < clan.dejavnosti.Count; n++)
```

```
            {
                for (int j = 0; j < FClanskiObrazec.cLBInteresne.Items.Count;
j++)
                    if (clan.dejavnosti[n].ToString() ==
                       FClanskiObrazec.cLBInteresne.Items[j].ToString())
                        FClanskiObrazec.cLBInteresne.SetItemChecked(j, true);
            }
            //članski obrazec sedaj odpremo kot dialog
            if (FClanskiObrazec.ShowDialog(this) == DialogResult.OK)
            {
                //Če je bil kliknjen gumb OK, AŽURIRAMO dataGridView1
                dataGridView1.CurrentRow.Cells[0].Value =
                     FClanskiObrazec.tBIme.Text;
                dataGridView1.CurrentRow.Cells[1].Value =
                     FClanskiObrazec.tBPriimek.Text;
                dataGridView1.CurrentRow.Cells[2].Value =
                     FClanskiObrazec.cBFunkcija.Text;
                dataGridView1.CurrentRow.Cells[3].Value =
                     FClanskiObrazec.dTPRojen.Value;
                if (FClanskiObrazec.radioButton1.Checked == true)
                    dataGridView1.CurrentRow.Cells[4].Value = 'Z';
                else dataGridView1.CurrentRow.Cells[4].Value = 'M';
                if (FClanskiObrazec.cBKadilec.Checked == true)
                    dataGridView1.CurrentRow.Cells[5].Value = true;
                else dataGridView1.CurrentRow.Cells[5].Value = false;
                //ustvarimo nov objekt za dejavnosti
                clan.dejavnosti = new ArrayList();
                //v objekt dodamo izbrane dejavnosti iz članskega obrazca
                for (int i = 0; i <
FClanskiObrazec.cLBInteresne.CheckedItems.Count; i++)
clan.dejavnosti.Add(FClanskiObrazec.cLBInteresne.CheckedItems[i]);
                dataGridView1.CurrentRow.Cells[6].Value = clan.dejavnosti;
            }
            /*sprostimo pomnilnik; če tega ne naredimo, bo to za nas enkrat
              kasneje naredil "smetar" - GarbageCollector  */
            FClanskiObrazec.Dispose();
        }
    }
    // odzivna metoda za brisanje izbrane vrstice
    private void toolStripButton2_Click(object sender, EventArgs e)
    {
        if (MessageBox.Show("Brišem izbrano vrstico! ","Brisanje vrstice"
,MessageBoxButtons.YesNo,MessageBoxIcon.Question)==DialogResult.Yes)
            dataGridView1.Rows.Remove(dataGridView1.CurrentRow);
    }
    // pred zaprtjem obrazca poskrbimo za shranjevanje vseh podatkov
    private void Fazuriraj_FormClosed(object sender, FormClosedEventArgs e)
    {
        /*klic metode za shranjevanje vsebine gradnika DataGridView v
          datoteko*/
```

```
            shraniVDatoteko();
        }
        /*naša lastna metoda, ki poskrbi za shranjevanje vsebine gradnika
          dataGridView1 v datoteko*/
        private void shraniVDatoteko()
        {
            try
            {
                string imeDatoteke = "SDTriglav.txt"; //določimo ime datoteke
                //ustvarimo novo datoteko, oz. odpremo obstoječo!
                Using (StreamWriter pisi=File.CreateText(imeDatoteke))
                {
                    /*zanka se izvede tolikokrat, kot je vrstic gradnika
                      dataGridView1 */
                    for (int i = 0; i < dataGridView1.Rows.Count; i++)
                    {   /*vsebine celic pridobimo v našem primeru s pomočjo
                          indeksa stolpca (začetni stolpec ima indeks 0!)
                          ločilo med posameznimi podatki v datoteki bo znak |*/
                      pisi.Write(dataGridView1.Rows[i].Cells[0].Value.ToString()
                      + "|" +
                      dataGridView1.Rows[i].Cells[1].Value.ToString() + "|" +
                      dataGridView1.Rows[i].Cells[2].Value.ToString() + "|" +
                      dataGridView1.Rows[i].Cells[3].Value.ToString() + "|" +
                      dataGridView1.Rows[i].Cells[4].Value.ToString() + "|" +
                      dataGridView1.Rows[i].Cells[5].Value.ToString());
                        /*objekt razreda clan potrebujemo zato, da vanj
                          prenesemo izbrane dejavnosti iz članskega obrazca*/
                        Clan clan = new Clan();
                        /*celico z indeksom 6, v kateri je zbirka dejavnosti,
                          prenesemo v objekt clan*/
                        clan.dejavnosti =
(ArrayList)dataGridView1.Rows[i].Cells[6].Value;
                        for (int n = 0; n < clan.dejavnosti.Count; n++)
                            pisi.Write("|" + clan.dejavnosti[n].ToString());
                        //dejavnosti po vrsti pišemo v datoteko
                        pisi.WriteLine();
                    }
                }
            }
            catch
            { MessageBox.Show("Napaka pri zapisovanju v datoteko! "); }
        }

        private void ZapisVDatoteko(object sender, EventArgs e)
        {
             /*klic lastne metode za shranjevanje vsebine gradnika
               dataGridView1 v tekstovno datoteko*/
             shraniVDatoteko();
        }
        private void SteviloKadilcev_Click(object sender, EventArgs e)
```

```
    {
        int skupaj = 0;
        for (int i = 0; i < dataGridView1.Rows.Count; i++)
        {
            if
(Convert.ToBoolean(dataGridView1.Rows[i].Cells["Kadilec"].Value))
                skupaj++;
        }
        MessageBox.Show("Skupno število kadilcev!: " + skupaj, "Kadilci! ");
    }

    private void steviloMladoletnikov_Click(object sender, EventArgs e)
    {
        int skupaj = 0;
        for (int i = 0; i < dataGridView1.Rows.Count; i++)
        {
            DateTime datum =
Convert.ToDateTime(dataGridView1.Rows[i].Cells["Rojstvo"].Value);
            if (datum.Year < DateTime.Now.Year - 18)
                skupaj++;
        }
        MessageBox.Show("Skupno število mladoletnikov v datoteki (osebe
mlajše od 18 let): "+skupaj, "Mladoletniki! ");
    }
}
```

Dodali smo še dve odzivni metodi za obdelavo vrstic in celic gradnika *DataGridView*. Ugotovili smo skupno število kadilcev in število mladoletnih članov.

Ostanejo nam še odzivne metode glavnega obrazca. V spodnjem levem kotu glavnega obrazca bo ves čas prikazan trenutni čas. Potrebujemo gradnik tipa *Timer*, ki mu napišemo odzivno metodo dogodka *Tick*.

```
public partial class FGlavni : Form
{
    public FGlavni()
    {
        InitializeComponent();
    }

    //v spodnjem levem kotu obrazca bo ves čas prikazan trenutni čas
    private void timer1_Tick(object sender, EventArgs e)
    {
        toolStripStatusLabel1.Text = DateTime.Now.ToLongTimeString();
        /*lahko tudi  ->  statusStrip1.Items[0].Text =
          DateTime.Now.ToLongTimeString();*/
    }
    //dodajanje novega člana: ustvarimo nov objekt članskega obrazca
    private void dodajaToolStripMenuItem_Click(object sender, EventArgs e)
```

```
    {
        //generiramo nov članski obrazec
        FClanskiObrazec FClanskiObrazec = new FClanskiObrazec();
        //članski obrazec napolnimo z začetnimi vrednostmi
        FClanskiObrazec.tBIme.Text = "";
        FClanskiObrazec.tBPriimek.Text = "";
        FClanskiObrazec.cBFunkcija.Text = "Prvi";
        FClanskiObrazec.dTPRojen.Value = DateTime.Today;
        FClanskiObrazec.cBKadilec.Checked = false;
        FClanskiObrazec.radioButton1.Checked = true;

        if (FClanskiObrazec.ShowDialog(this) == DialogResult.OK)
        {
            Clan Nov = new Clan(); //nov objekt izpeljan iz razreda Clan
            /*vsi gradniki na obrazcu Clan so public, zato imamo do njih
              neposreden dostop*/
            Nov.ime = FClanskiObrazec.tBIme.Text;
            Nov.priimek = FClanskiObrazec.tBPriimek.Text;
            Nov.funkcija = FClanskiObrazec.cBFunkcija.Text;
            Nov.rojstvo = FClanskiObrazec.dTPRojen.Value;
            if (FClanskiObrazec.radioButton1.Checked == true)
                Nov.spol = 'Z';//Z pomeni ženski spol
            else Nov.spol = 'M';
            if (FClanskiObrazec.cBKadilec.Checked == true)
                Nov.kadilec = true;
            else Nov.kadilec = false;
            //ustvarimo nov objekt za dejavnosti
            Nov.dejavnosti = new ArrayList();
            //v objekt Nov dodamo vse izbrane dejavnosti članskega obrazca
            for (int
i=0;i<FClanskiObrazec.cLBInteresne.CheckedItems.Count;i++)
            {
Nov.dejavnosti.Add(FClanskiObrazec.cLBInteresne.CheckedItems[i]);
            }
            //novega člana takoj dodamo tudi v datoteko vseh članov
            string imeDatoteke = "SDTriglav.txt"; //določimo ime datoteke
            StreamWriter datoteka;
            //ustvarimo novo datoteko, oz. odpremo obstoječo!
            datoteka = File.AppendText(imeDatoteke);
            datoteka.Write(Nov.ime + "|" + Nov.priimek + "|" + Nov.funkcija +
"|" + Nov.rojstvo + "|" + Nov.spol + "|" + Nov.kadilec);
            for (int j = 0; j < Nov.dejavnosti.Count; j++)
                datoteka.Write("|" + Nov.dejavnosti[j]);
            datoteka.WriteLine();
            datoteka.Close();
        }
        FClanskiObrazec.Dispose(); //sprostimo pomnilnik
    }

    //odzivna metoda za prikaz članstva
```

```
    private void ažurirajToolStripMenuItem_Click(object sender,EventArgs e)
    {
        /*ustvarimo obrazec za prikaz vsebine datoteke (ta je prikazana v
          gradniku DataGridView)*/
        FPregled Pregled = new FPregled();
        Pregled.ShowDialog();
    }
    //odzivna metoda za uporabnikovo razvedrilo – Tri v vrsto
    private void trivVrstoToolStripMenuItem_Click(object sender,EventArgs e)
    {
        /*do razreda TriVVrsto dostopamo preko imena imenskega prostora, ki
          je v tem primeru tudi TriVVrsto*/
        TriVVrsto.FTriVVrsto igra = new TriVVrsto.FTriVVrsto();
        igra.Show();
    }
    //še prikaz vizitke z osnovnimi podatki o programu
    private void vizitkaToolStripMenuItem_Click(object sender, EventArgs e)
    {
        new AboutBox1().ShowDialog();//okno z lastnostmi projekta
    }

    //odzivna metoda za zaključek programa
    private void konecToolStripMenuItem_Click(object sender, EventArgs e)
    {
        if (MessageBox.Show("Zaključek programa?", "KONEC",
MessageBoxButtons.OKCancel, MessageBoxIcon.Question) == DialogResult.OK)
            Application.Exit();
    }
}
```

MDI – Multi Document Interface (vmesnik z več dokumenti)

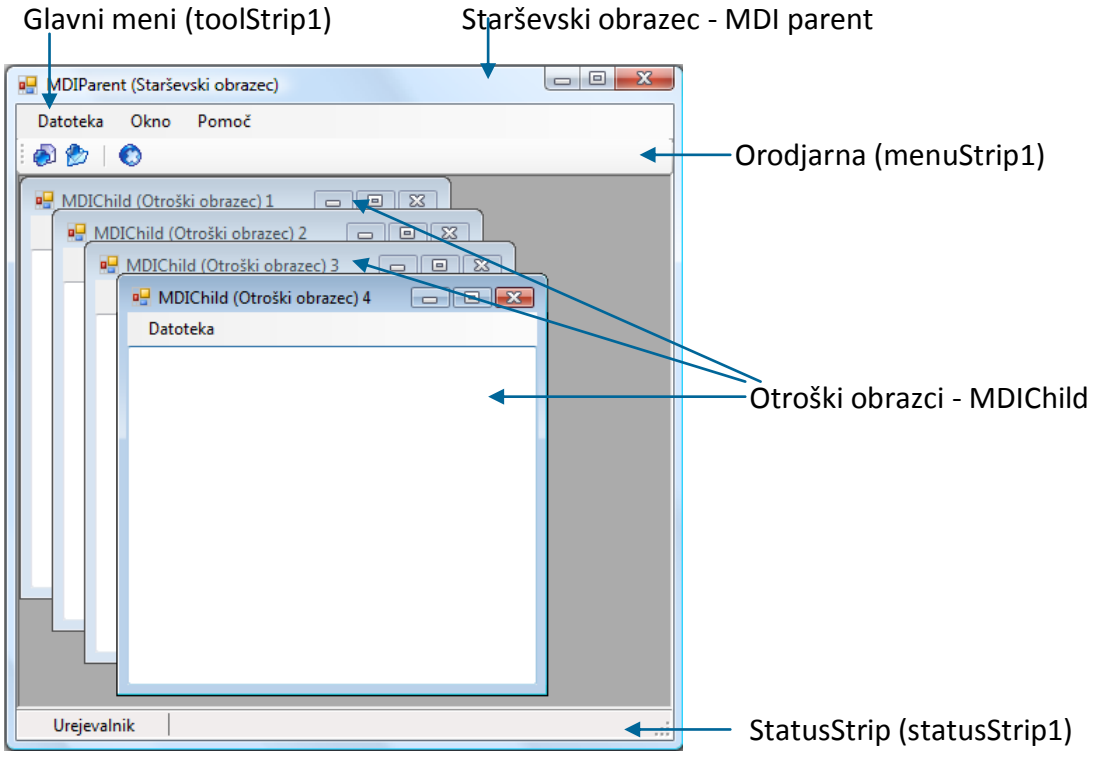
Zahtevnejši projekti navadno vsebujejo več obrazcev, saj z enim samim obrazcem uporabniku ne moremo ponuditi vseh potrebnih rešitev. Pri oblikovanju uporabniškega vmesnika in povezovanju obrazcev pa ločimo:

- Povezovanje obrazcev pri vmesniku *SDI* (*Single Document Interface* – vmesnik z enim samim dokumentom). Na tak način smo obrazce povezovali v vseh dosedanjih projektih. Izraz *SDI* ni najbolj posrečen, saj smo tudi pri dosedanjih obrazcih lahko imeli znotraj enega okna odprtih več dokumentov. *SDI* bi bolje prevedli kot *vmesnik enakovrednih oken*.

- Povezovanje obrazcev pri vmesniku *MDI* (*Multi Document Interface* – vmesnik z več dokumenti). *MDI* je vrsta uporabniškega vmesnika, kjer si okna s stališča programerja (in tudi uporabnika) niso enakovredna.

MDI uporabniški vmesnik navadno vsebuje natanko en obrazec tipa *MDI* (lastnost obrazca *IsMdiContainer* nastavimo na *true*). Temu obrazcu pravimo tudi *MDI parent* ali *starševski obrazec*. V delujočem programu lahko uporabnik odpre poljubno število enakovrednih podobrazcev in ima v vsakem od njih drugačno vsebino – tem obrazcem pa pravimo *MDI child* ali *otroški obrazci*. Primer *MDI* uporabniškega vmesnika so na primer urejevalniki besedil, v katerih imamo lahko odprtih več oken z različnimi besedili, pri čemer imamo dostop do skupnih urejevalnih orodij.

Vsebniški odnos med obrazci se vzpostavi šele ob zagonu programa. Posledica tega je, da se otroški obrazci (podobrazci) postavijo znotraj glavnega obrazca, ki je tipa *MDI*. Med izvajanjem programa vsebovanih oken ne moremo postaviti izven glavnega okna. Pri takem poskusu dobi glavno okno na robovih drsnike.

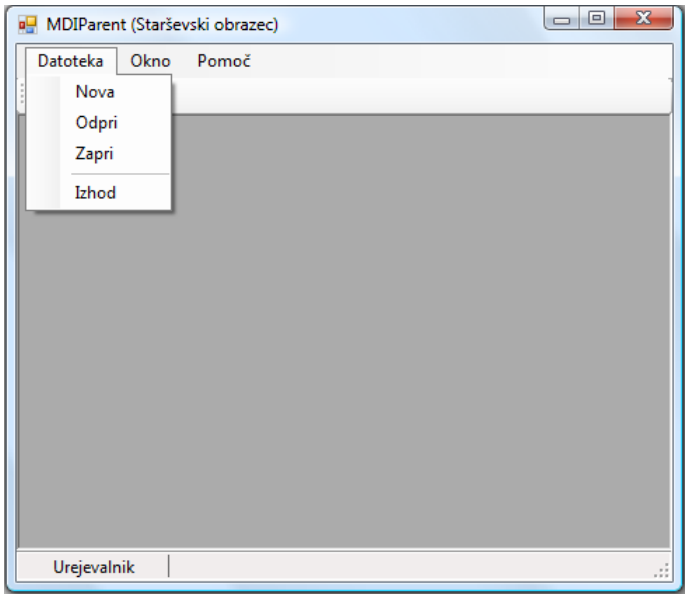


Slika 20: *MDI* obrazec in otroška okna.

Na glavni obrazec (*MDI parent*) navadno postavimo glavni meni, statusno vrstico in orodjarno. Vsi meniji, ki so na voljo otroškim obrazcem, morajo biti na starševskem obrazcu. Če namreč uporabnik preklaplja med otroškimi obrazci, se mora glavni meni nanašati na izbrani otroški obrazec. Ker lahko uporabnik odpre poljubno število otroških obrazcev, vseh obrazcev za vsa

podokna ne moremo izdelati vnaprej. Izoblikujemo le enega, ki služi kot vzorec za izdelavo, dejanske podobrazce pa ustvarimo dinamično.

Za vajo oblikujmo glavni obrazec npr. takole:



Slika 21: Starševski obrazec - *MDI Parent*.

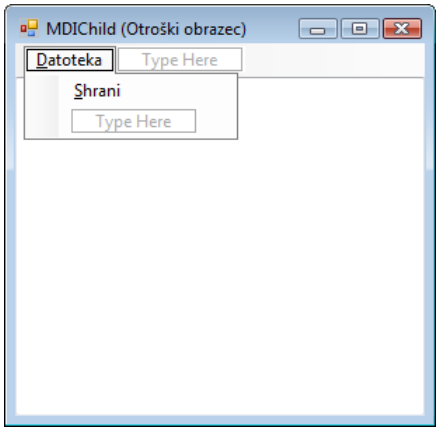
Obrazcu naj bo ime *fMDIParent*, lastnost *IsMdiContainer* postavimo na *true***.** Na ta način smo obrazec označili kot starševski obrazec, ki bo vseboval otroške obrazce. V glavnem meniju naredimo tri možnosti (*Datoteka*, *Okno* in *Pomoč*), postavka *Datoteka* ima tri možnosti (*Nova*, *Zapri* in *Izhod*), postavka *Okno* pa ravno tako tri možnosti (*Kaskade*, *Horizontalna razporeditev* in *Vertikalna razporeditev*).

V meniju *Datoteka* smo namenoma izpustili opcijo *Shrani*. Le-to bomo zapisali na otroškem obrazcu, nato pa s pomočjo zlivanja menijev (*MergeAction*) dosegli, da se bo po odprtju otroškega obrazca v meniju pojavila še ta možnost. V ta namen lastnost *MergeAction* možnosti *Datoteka* nastavimo na *MatchOnly*, vse ostale pa pustimo privzete (*Append*).

Ustvarimo še vzorec otroškega obrazca: meni *Project*, nato *Add Windows Form …* Za ime obrazca zapišimo *MDIChild.* Nanj postavimo meni z eno samo postavko *Datoteka* (lastnost *MergeAction* nastavimo na *MatchOnly*) in pod njo postavko *Shrani* (*MergeAction* nastavimo na *Insert*, *MergeIndex* pa npr. na 2).

Na obrazec dodajmo gradnik *TextBox (*ime naj bo *editData)*, ki mu lastnost *MultiLine* nastavimo na *True*, lastnost *Dock* pa na *Fill*.

Ker bomo potrebovali še dialog za shranjevanje datoteke, na obrazec dodajmo še gradnik *SaveFileDialog*.



Slika 22: Otroški obrazec - *MDI Child*.

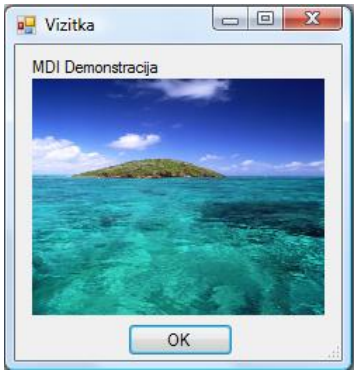
V starejših verzijah *Visual C#* se je za oblikovanje menija uporabljal gradnik *MainMenu*, v novejših pa se uporablja gradnik *MenuStrip* (še vedno pa lahko uporabljamo tudi gradnik *MainMenu*). Največja razlika med njima je nastavitev ustreznih lastnosti glede zlivanja menijev na starševskem in na otroških obrazcih. Pri gradniku *MainMenu* dosežemo zlivanje s pomočjo nastavitev lastnosti *MergeType* in *MergeOrder*, pri gradniku *MenuStrip* pa s pomočjo lastnosti *MergeAction*. Če npr. želimo, da meni na otroškem obrazcu nadomesti meni na glavnem obrazcu, nastavimo lastnost *MergeAction* na *Replace*.

V odzivni metodi za shranjevanje vsebine *TextBox*-a v datoteko, uporabimo kar metodo *WriteAllLines*.

```
public partial class MDIChild : Form
{
    public MDIChild()
    {
        InitializeComponent();
    }
    //odzivna metoda za shranjevanje vsebine otroškega okna
    private void shraniItem_Click(object sender, EventArgs e)
    {
        if (saveFileDialog.ShowDialog() == DialogResult.OK)
        {
            //ime datoteke izberemo s pomočjo SaveFileDialoga
            string datoteka = saveFileDialog.FileName;

              /*v izbrano datoteko zapišemo celotno vsebino gradnika TextBox
              (kodiranje je UTF8)*/
            File.WriteAllLines(datoteka,editData.Lines,Encoding.UTF8);
        }
        else MessageBox.Show("Podatki niso shranjeni!");
    }
}
```

Obrazec *MDIChild* sedaj shranimo. Uporabnik bo lahko odprl poljubno število primerkov (*instanc*) tega obrazca.

Dodajmo še obrazec *FVizitka*, ki naj bi bil namenjen pomoči, v bistvu pa bo to modalno okno s sliko. V našem primeru bo v oknu le slika in ime projekta, iz glavnega menija pa ga bomo odprli z metodo *Show*.



Slika 23: Modalno okno s pomočjo!

Odprimo sedaj glavni obrazec *fMDIParent***.** Ustvarimo in inicializirajmo še celoštevilsko spremenljivko *childCount*, s pomočjo katere bomo sledili, koliko otroških oken je že odprtih. Deklaracijo zapišimo kamorkoli znotraj razreda *fMDIParent* (pogled *Code View*), najbolje kar takoj na začetku. Zapišimo še odzivne metode. Za odpiranje nove datoteke ustvarimo novo otroško okno, ki mu določimo roditelja, zaporedno številko in napis na vrhu oknu.

```
public partial class fMDIParent : Form
{
    private int childCount = 0;//števec odprtih oken
    public fMDIParent()
    {
        InitializeComponent();
    }
   //odzivna metoda za odpiranje novega otroškega okna
    private void novaItem_Click(object sender, EventArgs e)
    {
        MDIChild childForm = new MDIChild();//Nov objekt obrazca MDIChild
        //roditelj otroškega okna je obrazec fMDIParent
        childForm.MdiParent = this;
        childCount++;//povečamo število otroških oken
        childForm.Text = childForm.Text + " " + childCount;//napis na oknu
        childForm.Show();//prikaz otroškega okna
    }
}
```

Za odpiranje obstoječe tekstovne datoteke uporabimo *OpenFileDialog*. Nastaviti mu moramo filter za prikaz le tekstovnih datotek. Ker bomo brali celotno vsebino datoteke naekrat, uporabimo metodo *ReadAllLines*. Za prikaz vsebine ustvarimo še novo otroško okno!

```
//odzivna metoda za odpiranje obstoječe tekstovne datoteke
private void toolStripMenuItem2_Click(object sender, EventArgs e)
{
    //odpiranje obstoječe tekstovne datoteke
    openFileDialog1.Filter = "Tekstovne datoteke|*.txt";
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        string datoteka = openFileDialog1.FileName;
        MDIChild childForm=new MDIChild();//Nov objekt obrazca MDIChild
        childForm.MdiParent = this;
        childCount++;
        childForm.Text = childForm.Text + " " + childCount;
        /*preberemo celo datoteko naenkrat in vsebino zapišemo v
         TextBox z imenom editData*/
        childForm.editData.Lines = File.ReadAllLines(datoteka,
Encoding.UTF8);
        childForm.Show();
    }
}
```

Tako kot katerokoli okno, tudi otroško okno zapremo z metodo *Close*. Pred zapiranjem še preverimo, če je kako otroško okno sploh še odrto. Tule je ustrezna odzivna metoda menijske vrstice za zapiranje okna.

```
private void zapriItem_Click(object sender, EventArgs e)
{
    /*Metoda  ActiveMdiChild vrne podatek o tem, katero otroško okno je
      trenutno aktivno, da ga bomo zaprli. Če še nobeno okno ni odprto,
      metoda ActiveMdiChild vrne vrednost null*/
    if (this.ActiveMdiChild != null)
        this.ActiveMdiChild.Close();
}
```

Pri delu z otroškimi okni imamo na voljo nekatere metode, s katerimi lahko odprta okna razporedimo v kaskadah, horizontalno ali pa vertikalno.

```
private void cascadeItem_Click(object sender, EventArgs e)
{
    //otroška okna razporedimo v kaskade
    this.LayoutMdi(MdiLayout.Cascade);
}

private void horizontalItem_Click(object sender, EventArgs e)
{
    //otroška okna razporedimo drugega ob drugem
```

```
    this.LayoutMdi(MdiLayout.TileHorizontal);
}

private void verticalItem_Click(object sender, EventArgs e)
{
    //otroška okna razporedimo eno pod drugim
    this.LayoutMdi(MdiLayout.TileVertical);
}
```

Napišimo še odzivni metodi za prikaz pomoči in za zapiranje programa.

```
private void aboutItem_Click(object sender, EventArgs e)
{
    //odpiranje modalnega okna s pomočjo
    fVizitka aboutDialog = new fVizitka();
    aboutDialog.ShowDialog();
}
//zapiranje starševskega okna in s tem tudi celega projekta
private void izhodItem_Click(object sender, EventArgs e)
{
    this.Close();
}
```

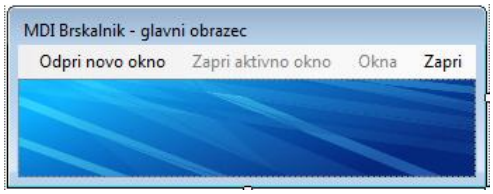
Brskalnik

Naredimo *MDI* projekt, ki ga bomo poimenovali *Brskalnik*. Naš brskalnik bo vseboval glavno okno iz katerega bomo lahko odprli poljubno število oken za brskanje po spletu. Glavno okno bo starševski obrazec, ostala okna bodo otroška. Starševski obrazec bo vseboval le vrstico z menijem, otroški obrazci pa domačo spletno stran, iskalnik, ter seznam priljubljenih spletnih strani. S pomočjo lebdečega menija bo poljubno spletno stran možno dodati med priljubljene spletne strani. Na otroških obrazcih želimo preverjali trenutno število odprtih otroških obrazcev, zato moramo imeti dostop do števila trenutno odprtih otroških obrazcev. Prav tako želimo imeto dostop do gradnikov *MenuStrip* in *PictureBox* na glavnem obrazcu. Tema dvema gradnikoma zato nastavimo lastnost *Modifiers* nastavimo na *Public*. Spremeniti moramo tudi datoteko *Program.cs*, kar smo spoznali že v poglavju *Dostop do polj, metod in gradnikov nadrejenega razreda*.

```
static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    public static FGlavni GlavniObrazec; //Najava glavnega obrazca
    [STAThread]

    static void Main()
```

```
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        GlavniObrazec=new FGlavni();//nov objekt izpeljan iz FGlavni
        Application.Run(GlavniObrazec); //Odpiranje glavnega obrazca
    }
}
```

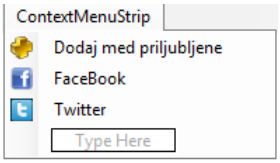
Glavnemu obrazcu lastnost *IsMDIContainer* nastavimo na *True*.



Slika 24: Glavni obrazec *MDI* aplikacije.

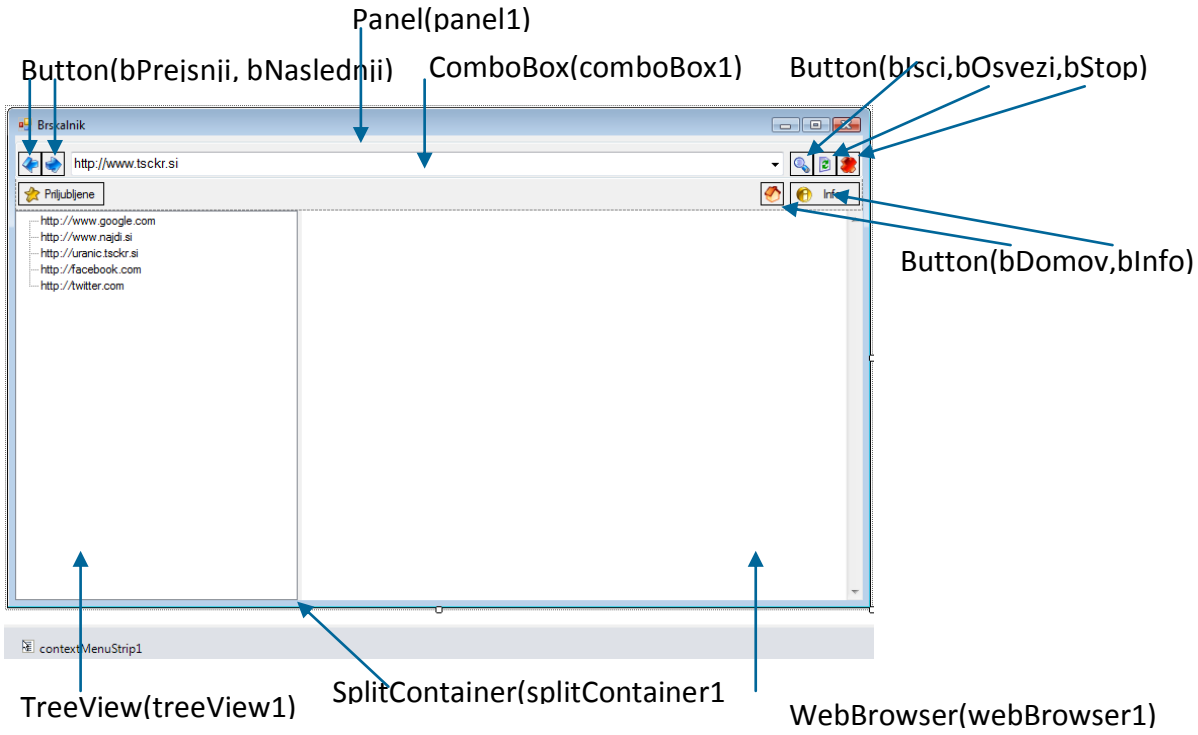
Otroški obrazec poimenujemo *FOtroski*. Nanj postavimo najprej Panel (*Dock=Top*), nanj pa *ComboBox* za vnos nove spletne strani in nekaj gumbov za navigacijo. Ostali del obrazca zapolnimo z gradnikom *SplitContainer* (*Dock=Fill*). Na levi panel tega gradnika postavimo gradnik *TreeView*, na katerem bomo lahko prikazali priljubljene spletne strani, na desni panel pa gradnik *WebBrowser*, ki zapolni celotno desno stran gradnika *SplitContainer*. V gradnik *TreeView* zapišimo nekaj naslovov spletnih strani takole: gradnik najprej izberemo, nato pa v oknu *Properties* kliknemo lastnost *Nodes*, da se prikaže okno *TreeNode Editor*. S klikom na gumb *Add* dodamo nekaj novih postavk – naslovov spletnih strani. Vsaki postavki določimo le lastnost *Text* (npr. http://www.google.com). Ostale lastnosti pustimo privzete.

Na obrazec postavimo še lebdeči meni (*ContextMenuStrip*). Gradnik dobi ime *contextMenuStrip1*, v njem ustvarimo le tri opcije:



Slika 25: Lebdeči meni na otroškem obrazcu!

Ta lebdeči meni priredimo gradniku *webBrowser1*, ki je že na obrazcu (za lastnost ContextMenuStrip tega gradnika izberemo *contextMenuStrip1*).



Slika 26: Otroški obrazec - brskalnik.

Pred odpiranjem prvega otroškega obrazca tudi poskrbimo, da se glavni obrazec razširi čez celoten zaslon. To naredimo s pomočjo lastnosti *ActiveMdiChild*, ki hrani ime trenutno aktivnega otroškega obrazca. Če noben otroški obrazec ni odprt, ima lastnost *ActiveMdiChild* vrednost *null*. S to lastnostjo si pomagamo tudi v odzivni metodi za zapiranje pojekta.

V meniju glavnega obrazca zapišemo tudi odzivne metode za razporejanje oken. Spoznali smo jih v prejšnjem primeru.

```
Public partial class FGlavni : Form
{
    public int otroskih = 0;//števec aktivnih otroških obrazcev
    public FGlavni()
    {
        InitializeComponent();
    }
    //odzivna metoda za odpiranje novega otroškega okna
    private void odpriNovoOknoToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        if (this.ActiveMdiChild == null)//če nobeno otroško okno odprto
        {
            //prikažemo srednji opciji glavnega menija
            zapriAktivnoOknoToolStripMenuItem.Enabled = true;
            oknaToolStripMenuItem.Enabled = true;
```

```
            pictureBox1.Hide();//skrijem sliko
            //okno razširimo čez celoten zaslon
            this.WindowState=FormWindowState.Maximized;
        }
        //odpremo otroški obrazec
        Fotroski otroski = new Fotroski();
        otroski.MdiParent = this;//njegov starš je objekt obrazca FGlavni
        otroskih++;//povečanm število aktivnih otroških obrazcev
        otroski.Show();
    }

    private void zapriAktivnoOknoToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        //če je še kakšno otroško okno aktivno
        if (this.ActiveMdiChild != null)
            this.ActiveMdiChild.Close();//ga zaprem
        /*če ni več nobenega aktivnega otroškega okna, pomanjšamo glavni
          Obrazec*/
        if (this.ActiveMdiChild == null)
        {
            zapriAktivnoOknoToolStripMenuItem.Enabled = false;
            oknaToolStripMenuItem.Enabled = false;
            pictureBox1.Show();
            this.WindowState = FormWindowState.Normal;
            this.Width = 350;
            this.Height = 130;
        }
    }

    private void kaskadeToolStripMenuItem_Click(object sender, EventArgs e)
    {
        //odprta otroška okna postavim v kaskade
        this.LayoutMdi(MdiLayout.Cascade);
    }

    private void drugoObDrugemToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        //odprta otroška okna postavim drugega ob drugem
        this.LayoutMdi(MdiLayout.TileHorizontal);
    }

    private void zapriToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Close();//zaprem projekt
    }
}
```

Več dela bomo imeli s pisanjem odzivnih metod otroškega obrazca. V njegovem konstruktorju najprej poskrbimo za prikaz privzete spletne strani. Njen naslov smo zapisali v lastnost *Items* gradnika *ComboBox* na vrhu obrazca.

```
public partial class FOtroski : Form
{
    public FOtroski()
    {
        InitializeComponent();
        //v brskalniku prikažemo stran, ki je v ComboBox-u zapisana prva
        string naslov = comboBox1.Text;
        webBrowser1.Navigate(new Uri(naslov));
        //določimo velikost otroškega obrazca
        this.Width = 1100;
        this.Height = 700;
        //skrijemo gumb Priljubljene
        splitContainer1.Panel1MinSize = 0;
        splitContainer1.SplitterDistance = 0;
    }
}
```

Uporabnik bo lahko v vnosno polje gradnika *ComboBox* vnesel poljubno spletno stran. Navigacijo na to spletno stran poskucimo izvesti s pritiskom na tikpo <*Enter*> ali pa s klikom na gumb *bIsci*. Uporabili bomo metodo *Navigate* objekta *webBrowser1*. Parameter te metode je objekt tipa *Uri* (*Uniform resoutce identifier*), ki ga ustvarimo glede na vneseno spletno stran.

```
//poskus navigacije na vpisano spletno stran
private void bIsci_Click(object sender, EventArgs e)
{
    try
    {
        //skušamo odpreti želeno spletno stran
        string naslov = comboBox1.Text;
        //metoda Navigate ima za parameter objekt tipa Uri
        webBrowser1.Navigate(new Uri(naslov));
    }
    catch
    { MessageBox.Show("Napaka, ali pa stran ne obstaja! ", "POZOR!", 0,
MessageBoxIcon.Warning); }
}
//odzivna metoda se izvede, če je v ComboBox-u kliknjena tipka Enter
private void comboBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    try
    {
        if (e.KeyChar == (char)13)
            webBrowser1.Navigate(comboBox1.Text);
    }
    catch
```

```
    { MessageBox.Show("Napaka, ali pa stran ne obstaja!", "POZOR! ", 0,
MessageBoxIcon.Warning); }
}
```

Če spletna stran obstaja, se prične nalagati v naše okno. Ko je stran prikazana v celoti, osvežimo prikazano vrstico gradnika *ComboBox*. Nato še preverimo, ali ta stran v seznamu že obstaja. Če gre za novo stran, jo dodamo. Odzivna metoda je prirejena dogodku *DocumentCompleted* objekta *webBrowser1*.

```
//odzivna metoda, ki se izvede ko je spletna stran uspešno prikazana
private void webBrowser1_DocumentCompleted(object sender,
WebBrowserDocumentCompletedEventArgs e)
{
    //ko je spletna stran prikazana v celoti, osvežimo ComboBox
    comboBox1.Text = webBrowser1.Document.Url.ToString();
   //če ta stran še ni bila obiskana, jo dodamo v ComboBox
    if (!comboBox1.Items.Contains(comboBox1.Text))
        comboBox1.Items.Add(comboBox1.Text);
}
```

Ob kliku na gumb *Priljubljene* se odpre vsebina gradnika *TreeView*, ki vsebuje vnaprej pripravljene naslove priljubljenih spletnih strani. Ob kliku na katerokoli od teh se naj izvede odzivna metoda, ki poskrbi za navigacijo na to stran.

```
//klik na gumb Priljubljene
private void bPriljubljene_Click(object sender, EventArgs e)
{
    /*če so priljubljene spletne strani skrite, jih prikažemo, sicer pa
       jih skrijemo*/
    if (splitContainer1.SplitterDistance == 0)
        splitContainer1.SplitterDistance = 250;
    else splitContainer1.SplitterDistance = 0;
}

//dvoklik na priljubljeno spletno stran
private void treeView1_NodeMouseDoubleClick(object sender,
TreeNodeMouseClickEventArgs e)
{
    try
    {
        comboBox1.Text = treeView1.Nodes[treeView1.SelectedNode.Index].Text;
        webBrowser1.Navigate(comboBox1.Text);
    }
    catch { }
}
```

Všečne spletne strani želimo dodajati med priljubljene. Zato napišimo odzivno metodo lebdečega menija, ki se odpre ob desnem kliku miške, ko je ta nad objektom *webBrowse1*. Novo

bližnjico do spletne strani bomo dodali kar v glavno vejo objekta t*reeView1*, ki se imenuje "*A*". Dodajmo še odzivni metodi za ostali dve možnosti lebdečega menija.

```
//odzivna metoda za dodajanje spletne strani v seznam priljubljenih
private void dodajMedPriljubljeneToolStripMenuItem_Click(object sender,
EventArgs e)
{
    /*v gradnik TreeView dodamo novo spletno stran. Dodamo jo v glavno vejo
      Objekta treeView1; glavna veja se imenuje "A"*/
    treeView1.Nodes.Add("A",webBrowser1.Url.ToString(),1);
}
//navigacija na spletno stran http://twitter.com
private void twitterToolStripMenuItem_Click(object sender, EventArgs e)
{
    comboBox1.Text = "http://twitter.com/";
    webBrowser1.Navigate(comboBox1.Text);
}
//navigacija na spletno stran http://facebook.com
private void faceBookToolStripMenuItem_Click(object sender, EventArgs e)
{
    comboBox1.Text = "http://facebook.com/";
    webBrowser1.Navigate(comboBox1.Text);
}
```

Pred zaprtjem otroškega obrazca zmanjšamo skupno število odprtih otroških oken. Če so zaprta vsa otroška okna, poskrbimo za začetne nastavitve glavnega obrazca.

```
//odzivna metoda, ki se izvede ob zapiranju otroškega obrazca
private void FOtroski_FormClosed(object sender, FormClosedEventArgs e)
{
    //zmanjšamo število aktivnih otroških oken
    MDIBrskalnik.Program.GlavniObrazec.otroskih--;
    //če je število otroških oken enako 0, pomanjšamo glavni obrazec
    if (MDIBrskalnik.Program.GlavniObrazec.otroskih == 0)
    {
MDIBrskalnik.Program.GlavniObrazec.zapriAktivnoOknoToolStripMenuItem.Enabled
= false;
        MDIBrskalnik.Program.GlavniObrazec.oknaToolStripMenuItem.Enabled =
            false;
        MDIBrskalnik.Program.GlavniObrazec.pictureBox1.Show();
        MDIBrskalnik.Program.GlavniObrazec.WindowState=
         FormWindowState.Normal;
        MDIBrskalnik.Program.GlavniObrazec.Width = 350;
        MDIBrskalnik.Program.GlavniObrazec.Height = 130;
    }
}
```

Na koncu dodajmo še nekaj odzivnih metod gumbov, ki so že na obrazcu. Z njimi bomo obogatili delovanje brskalnika.

```
//odzivna metoda za osveževanje brskalnika
private void bOsvezi_Click(object sender, EventArgs e)
{
    webBrowser1.Refresh();
}
//odzivna metoda za prekinitev navigacije brskalnika
private void bStop_Click(object sender, EventArgs e)
{
    webBrowser1.Stop();
}
//odzivna metoda za premik na prejšnjo stran
private void bPrejsnja_Click(object sender, EventArgs e)
{
    if (webBrowser1.GoBack())
    {
        comboBox1.ResetText();
    }
}
//odzivna metda za premik na naslednjo stran
private void bNaslednja_Click(object sender, EventArgs e)
{
    if (webBrowser1.GoForward())
    {
        comboBox1.ResetText();
        while (webBrowser1.IsBusy) { }
    }
}
//klik na gumb Domov
private void bDomov_Click(object sender, EventArgs e)
{
    comboBox1.Text = "http://www.tsckr.si";
    webBrowser1.Navigate(comboBox1.Text);
}
//klik na gumb bInfo
private void bInfo_Click(object sender, EventArgs e)
{
    string info = "Ob vsaki uspešni navigaciji se naslov shrani v
                   ComboBox\n";
    info+="Naslovi obiskanih spletnih strani so zapisani po abecedi.";
    info+="\nDesni klik na spletno stran prikaže lebdeči meni!";
    info += "\nOdpri Priljubjene in DVOKLIKNI ustrezno spletno stran";
    MessageBox.Show(info);
}
```

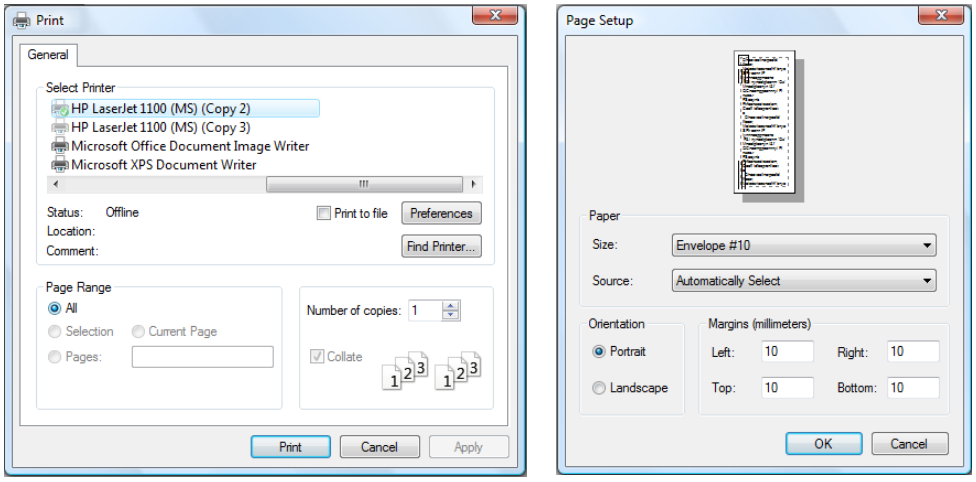
Tiskalnik

Gradnike, ki so v *Visual C#* namenjeni tiskalniškim poslom, najdemo v oknu *Toolbox* v skupini *Printing***.** Gradniki so večinoma nevizuelni. Ko jih postavimo na obrazec, se prikažejo v polju pod obrazcem. Tiskamo lahko besedila in slike, večino nastavitev za tiskanje pa moramo narediti programsko.

Za izbiro in nastavitev tiskalnika, ter za nastavitev strani izpisa sta na voljo dialoga

- *PrintDialog*: izbira tiskalnika, obseg tiskanja, število kopij, …
- *PageSetupDialog*: velikost papirja, izbira pladnja, orientacija tiskanja in robovi izpisa.

Oba dialoga odpremo z metodo *ShowDialog* in preverjamo vrednost *DialogResult*, ki jo dialoga vrneta. Izbiro tiskalnika (ali pa njegove nastavitve) potrdimo ali pa prekličemo s klikom na enega od modalnih gumbov na obrazcu. Za samostojno odpiranje *PageSetupDialog*-a moramo s pomočjo lastnosti *Document* najprej določiti dokument. Ta je tipa *PrintDocument*. Z njim določimo vsebino, ki jo želimo poslati tiskalniku in šele nato lahko v pogovornem oknu spreminjamo nastavitve za ta dokument.



Slika 27: Dialog za izbiro tiskalnika in dialog za nastavitev strani izpisa.

```
//odpiranje printDialog-a
if (printDialog1.ShowDialog()==DialogResult.OK)
    //kliknjen je bil gumb Print
//uporaba PageSetupDialoga
pageSetupDialog1.Document = printDocument1;
if (pageSetupDialog1.ShowDialog()==DialogResult.OK)
    //kliknjen je bil gumb OK
```

Učno gradivo je nastalo v okviru projekta Munus 2. Njegovo izdajo je omogočilo sofinanciranje Evropskega socialnega sklada Evropske unije in Ministrstva za šolstvo in šport.

Gradnik *PrintDocument* je namenjen neposrednemu tiskanju. Uporablja se za nastavitev lastnosti, s katerimi povemo kaj želimo tiskati. Z njegovo metodo *PrintPage* dokument nato tudi natisnemo. Pri tem uporabljamo metode in lastnosti razreda *PrintPageEventArgs*, s katerimi določimo kaj bomo tiskali. Tiskamo lahko besedilo (*Graphics.DrawString*), sliko (*Graphics.DrawImage*), geometrijski lik (*Graphics.DrawRectangle*), črto (*Graphics.DrawLine*), elipsa/krog (*Graphics.DrawEllipse*) …

Metode/Lastnosti	Razlaga
Graphics	Lastnost za tiskanje strani.
HasMorePages	Nastavitev (*true/false*) vrednosti s katero označimo, ali obstaja še dodatna stran za tiskanje.
MarginBounds	Podatki o pravokotnem področju namenjenem tiskanju znotraj tiskalnikove strani.
PageBounds	Podatki o pravokotnem območju, ki predstavlja celotno površino strani.
PageSetings	Nastavitve tekoče strani.

Tabela 2: Najpomembnejše lastnosti in metode razreda *PrintPageEventArgs*.

Vse, v zgornji tabeli naštete lastnosti, vsebujejo številne podatke, ki jih lahko uporabimo pri tiskanju. Predvsem metode za tiskanje imajo veliko število preobtežitev, v naslednjem primeru in vajah pa bodo prikazane le nekatere med njimi.

Gradnik PrintDocument navadno uporabljamo v tesni povezavi z gradnikom PrintDialog ali pa PrintPreviewDialog, s pomočjo katerih izberemo ustrezen tiskalnik, določimo del dokumenta, ki ga želimo natisniti, število kopij in še nekatere druge nastavitve.

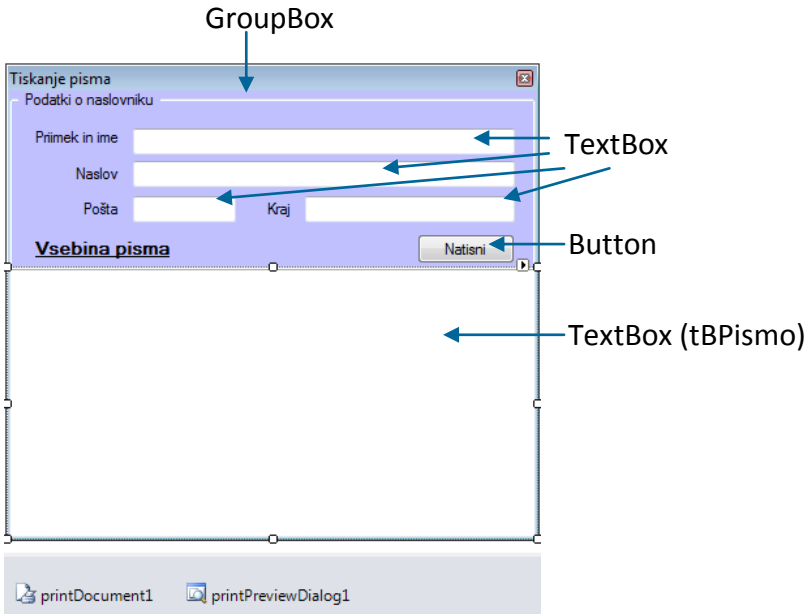
Tiskanje enostranskega besedila

Ustvarimo obrazec z gradniki za vnos podatkov o naslovniku in vsebino poljubnega pisma. Vnesene podatke bomo v primerni obliki natisnili na tiskalniku. Vsebina pisma naj zaenkrat ne presega ene strani.

Pri tiskanju vsebine pisma bomo, glede na izbrano velikost pisave, v vsaki vrstici natisnili največ 95 znakov. Če bo znakov v vrstici gradnika *TextBox* več kot 95, jo bomo razdelili na dve ali več vrstic. Zaradi enostavnosti pri tem seveda ne bomo pazili na pravilno deljenje besed. Uporabili bomo tudi gradnik *PrintPreviewDialog* za predogled tiskanja.

Najprej pripravimo obrazec za vnos uporabnikovih podatkov in vsebino pisma. Gradniki tipa *TextBox* naj imajo zaporedoma imena *tBIme*, *tBNaslov*, *tBPosta* in *tBKraj*. Tudi za vsebino pisma bomo uporabili gradnik tipa *TextBox*. Poimenujmo ga *tBPismo* in mu lastnost *MultiLine* nastavimo na *True*.

Na obrazcu je še gumb z napisom *Natisni*, za začetek tiskanja. Ob kliku na gumb bomo vsebino pisma prenesli v tabelo nizov z imenom *pismo*. Za delo s tiskalnikom potrebujemo še gradnik tipa *PrintDocument*, za predogled tiskanja pa *PrintPreviewDialog*.



Slika 28: Obrazec za tiskanje pisma.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    Font printFont; //pred tiskanjem bomo vsakič določili pisavo
    string strPrint;/*niz, v katerem bomo vsakič zapisali besedilo za
                      tiskanje*/
    string[] pismo; //vsebino pisma bomo pred tiskanjem shranili v tabelo
    //odzivna metoda gumba Natisni
    private void bNatisni_Click(object sender, EventArgs e)
    {
        pismo = tBPismo.Lines; /*vsebino pisma pred tiskanjem shranimo v
                                 tabelo*/
        /*določimo ime TISKARNIŠKEGA posla - POZOR: TO NI IME dokumenta, ki
          ga želimo natisniti!!*/
        printDocument1.DocumentName = "Tiskanje pisma";
```

```
        //povežemo dilaog za predogled z gradnikom za tiskanje
        printPreviewDialog1.Document = printDocument1;
        /*Pred dokončnim tiskanjem MORAMO definirati še dogodek PrintPage,
          kjer povemo, kaj bomo sploh tiskali*/
        printPreviewDialog1.ShowDialog();
}
```

Ključna odzivna metoda, kjer bomo pravzaprav povedali kaj želimo natisniti, je odzivna metoda dogodga *PrintPage*. Priredimo jo objektu *printDocument1*. Drugi parameter te odzivne metode je tipa *PrintPageEventArgs*. Preko tega objekta pridemo do osnovnih podatkov o nastavitvah tiskalnika, s katerim bomo tiskali. Z eno od metod razreda *Graphics*, ki jih ta objekt pozna, pa povemo kaj bomo tiskali. Metodi moramo za parameter poslati natančne podatke o tiskanju. Metoda DrawString npr. potrebuje niz, ki ga želimo natisniti, font, barvo pisave, ter natančno pozicijo začetka tiskanja.

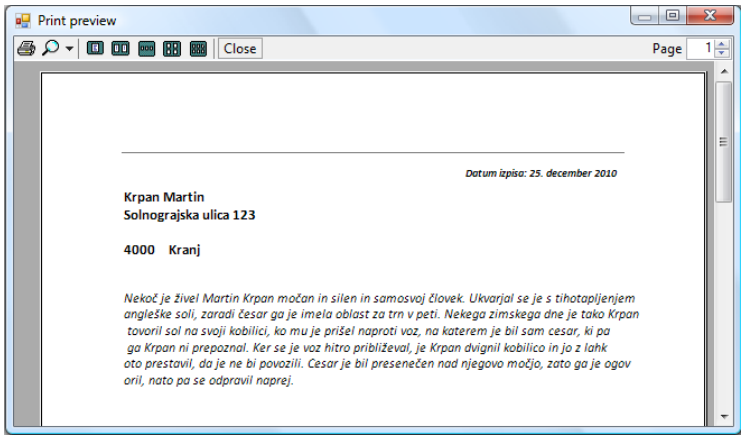
V odzivni metodi moramo poskrbeti tudi za obvestilo, da je tiskanje zaključeno. To storimo s pomočjo lastnosti *HasMorePages*, ki jo postavimo na *False* .

```
/*dogodek PrintPage gradnika printDocument1 – v njem zapišemo kaj in
  kako bomo tiskali*/
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    /*e.MarginBounds predstavlja pravokotno področje namenjeno tiskanju
    znotraj tiskalnikove strani*/
    float leviRob = e.MarginBounds.Left; //oddaljenost od levega roba
    float zgornjiRob = e.MarginBounds.Top;//oddaljenost od vrha strani
    float sirina = e.MarginBounds.Width; //širina izpisa
    float visina = e.MarginBounds.Height;//višina izpisa
    //font, ki ga bomo uporabili pri tiskanju črte in datuma izpisa
    printFont = new Font("Calibri", 10, FontStyle.Bold | FontStyle.Italic);
    //na vrhi strani za vajo natisnemo vodoravno črto debeline 2
    e.Graphics.DrawLine(new Pen(Color.Gray, 2), leviRob, zgornjiRob, leviRob
      + sirina, zgornjiRob);
    /*yPos pomeni oddaljenost (vrstice, ki jo želimo natisniti ) od
        vrha strani*/
    float yPos = zgornjiRob + printFont.GetHeight(e.Graphics);
    string datum = DateTime.Now.ToLongDateString();
    //pod zgornjo črto natisnemo datum izpisa
    e.Graphics.DrawString("Datum izpisa: " + datum, printFont, Brushes.Black,
       sirina-100, yPos, new StringFormat());
    //font, ki ga bomo uporabili pri tiskanju naslovnika
    printFont = new Font("Calibri", 13, FontStyle.Bold);
    yPos = zgornjiRob + 2*printFont.GetHeight(e.Graphics);
    //natisnemo ime in priimek
    e.Graphics.DrawString(tBIme.Text, printFont, Brushes.Black, leviRob,
      yPos, new StringFormat());
    yPos = yPos + printFont.GetHeight(e.Graphics);
```

```
    //natisnemo naslov
    e.Graphics.DrawString(tBNaslov.Text, printFont, Brushes.Black, leviRob,
       yPos, new StringFormat());
    yPos = yPos + 2 * printFont.GetHeight(e.Graphics);
    //natisnemo poštno številko in kraj
    e.Graphics.DrawString(tBPosta.Text+"    "+tBKraj.Text, printFont,
       Brushes.Black, leviRob, yPos, new StringFormat());

    //določimo še font za tiskanje vsebine pisma.
    printFont = new Font("Calibri",12, FontStyle.Regular | FontStyle.Italic);
    yPos = yPos + 2 * printFont.GetHeight(e.Graphics);
    int vrstic = pismo.Length;
    int znakov = 95;//največje število znakov v vrstici
    //natisnemo še vsebino pisma
    for (int i = 0; i < vrstic; i++)
    {
        string vrstica = pismo[i];
        do /*ponavljamo toliko časa, da je natisnjena celotna vrstica
                gradnika tBPismo*/
        {
            yPos = yPos + printFont.GetHeight(e.Graphics);
            string natisni;
            //če je v vrstici več kot 95 znakov jo razdelimo
            if (vrstica.Length > znakov)
            {
                natisni = vrstica.Substring(0, znakov);
                vrstica = vrstica.Substring(znakov, vrstica.Length - znakov);
                //tiskanje vrstice do 95 znakov
                e.Graphics.DrawString(natisni, printFont, Brushes.Black,
                   leviRob, yPos, new StringFormat());
            }
            else //če je znakov manj kot 95, jo natisnemo
            {
                e.Graphics.DrawString(vrstica, printFont, Brushes.Black,
                   leviRob, yPos, new StringFormat());
                break; //izhod iz for zanke
            }
        }
        while (vrstica.Length > 0);
    }

    //za vajo natisnemo še vodoravno črto na dnu pisma
    e.Graphics.DrawLine(new Pen(Color.Black, 1), leviRob,visina, leviRob +
        sirina, visina);
    e.HasMorePages = false;//tiskanje je zaključeno
}
```

Med tiskanjem smo morali ves čas paziti na oddaljenost od zgornjega roba. V odzivni metodi je zato skrbela spremenljivka *yPos*.

Predogled je pravzaprav vnaprej pripravljen obrazec. Na njem je prikazan izgled našega pisma, poleg tega pa še nekaj gradnikov: gumb za tiskanje, spustni seznam za nastavitev velikosti predogleda, gumbi za prikaz ene, dveh in več strani hkrati, gumb za zapiranje predogleda (predogled se zapre, tiskanje na tiskalniku se ne izvede) in gumb za prikaz določene številke strani.

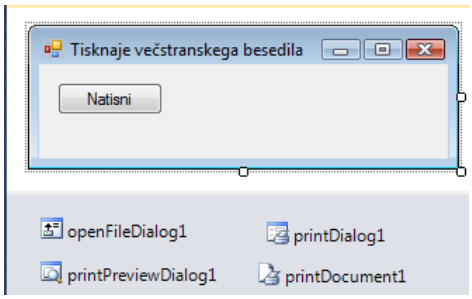


Slika 29: Predogled tiskanja.

Tiskanje besedila, ki obsega poljubno število strani

Pri tiskanju večstranskega besedila uporabimo metodo *MeasureString*, ki nastavi vse potrebne parametre za izpis strani na izbranem tiskalniku.

Pripravimo obrazec, na katerem bo en sam gumb, dodamo pa še gradnike *OpenFileDialog*, *PrintPreviewDialog*, *PrintDialog* in *PrintDocument*.



Slika 30: Obrazec za tiskanje večstranskega besedila.

Odzivno metodo priredimo dogodku *Click* gumba *bNatisni*, ki smo ga postavili na obrazec. S pomočjo *OpenFileDialog*-a izberemo poljubno tekstovno datoteko. Vsebino te datoteke preberemo in shranimo v spremenljivko tipa niz z imenom *strPrint*. Nato zaženemo predogled.

```
Font printFont; //pred tiskanjem bomo vsakič določili pisavo
string strPrint;//niz, v katerega bomo pisali besedilo za tiskanje
//odzivna metoda za tiskanje poljubnega števila strani: predogled
private void TiskanjeBesedila_Click(object sender, EventArgs e)
{
    //napis na pogovornem oknu OpenFileDialog
    openFileDialog1.Title = "Okno za iskanje tekstovne datoteke, ki jo želiš
natisniti!";
    //filter za prikaz tekstovnih datotek
    openFileDialog1.Filter = "Tekstovne datoteke|*.txt";
    //z OpenFile dialogom izberemo datoteko za tiskanje
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        //s PrintPreviewDialogom izbiremo tiskalnik
        if (printDialog1.ShowDialog() == DialogResult.OK)
        {
            /*nastavitve za tiskanje dokumenta naj bodo take, kot smo jih
              določili v PrintDialogu*/
            printDocument1.PrinterSettings = printDialog1.PrinterSettings;
            printDocument1.DocumentName = "Tiskanje dokumenta " +
               openFileDialog1.FileName;
            printPreviewDialog1.Document = printDocument1;
            //Vsebino datoteke shranimo (preberemo) v niz strPrint
            StreamReader beri = File.OpenText(openFileDialog1.FileName);
            //vsebino tekstovne datoteke shranimo v niz
            strPrint = beri.ReadToEnd();
            beri.Close();
            /*določimo še font za tiskanje. Spremenljivka printFont je
            deklarirana kot globalna spremenljivka*/
            printFont = new Font("Arial", 12, FontStyle.Bold |
              FontStyle.Italic);
            /*Pred dokončnim tiskanjem MORAMO definirati še dogodek
            PtintPage, kjer povemo, kaj bomo sploh tiskali*/
            printPreviewDialog1.ShowDialog();
        }
    }
}
```

Potrebujemo še odzivno metodo dogodka *PrintPage*, ki jo priredimo objektu *printDocument1*. V njej z metodo *DrawString* tiskamo vrstice. Po vsaki natisnjeni vrstici določimo besedilo naslednje vrstice. Če je tiskanje zaključeno, lastnost *HasMorePages* nastavimo na *False*, sicer pa na *True*.

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    int charCount = 0;  //charCount: število znakov v vrstici
    int lineCount = 0;  //lineCount: število vrstic na stran
    /*metoda MeasureString določi potrebne parametre za tiskanje posamezne
```

```
      strani*/
    e.Graphics.MeasureString(strPrint, printFont, e.MarginBounds.Size,
StringFormat.GenericTypographic, out charCount, out lineCount);
    //Natisnemo stran
    e.Graphics.DrawString(strPrint, printFont, Brushes.Black, e.MarginBounds,
StringFormat.GenericTypographic);
    //Odstranimo del niza, ki je bil že natisnjen
    strPrint = strPrint.Substring(charCount);
    //Preverimo, če je še kaj za natisniti
    if (strPrint.Length > 0)
        e.HasMorePages = true;
    else
        e.HasMorePages = false;
}
```

Tiskanje slik

Pripravimo obrazec tako kot v prejšnjem primeru. Za tiskanje slike bomo najprej ustvarili objekt tipa *Bitmap* in s pomočjo *OpenFileDialog*-a van prenesli ustrezno sliko.

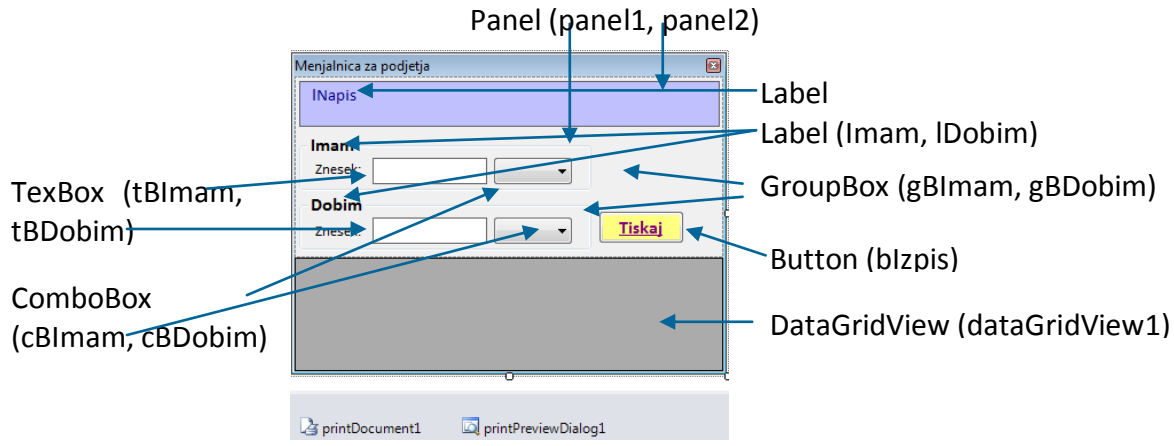
```
//Tiskanje slike: predogled
Font printFont; //pred tiskanjem bomo vsakič določili pisavo
string strPrint;//niz, v katerega bomo pisali besedilo za tiskanje
private void tiskanjeSlikePredogledToolStripMenuItem_Click(object sender,
EventArgs e)
{
    //napis na oknu
    openFileDialog1.Title= "Okno za iskanje slike, ki jo želiš natisniti!";
    //nastavitev filtra za ustrezne formate slik
    openFileDialog1.Filter = "Slike(*.jpg *.bmp *.jpeg *.gif *.png *.tiff
*.wmf)|*.jpg;*.bmp;*.jpeg;*.gif;*.png;*.tiff;*.wmf";
    //izberemo sliko
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {   //izberemo tiskalnik
        if (printDialog1.ShowDialog() == DialogResult.OK)
        {
            //določimo ime TISKARNIŠKEGA posla
            printDocument1.DocumentName = "Tiskanje SLIKE: " +
openFileDialog1.FileName;
            //Sliko izberemo z OpenFileDialog-om
            slika = new Bitmap(openFileDialog1.FileName, true);
            /*Pred dokončnim tiskanjem MORAMO definirati še dogodek
              PrintPage, kjer povemo, kaj bomo sploh tiskali*/
            printPreviewDialog1.Document = printDocument3;
            printPreviewDialog1.ShowDialog();
        }
    }
}
```

Objektu *printDocument1* priredimo še odzivno metodo dogodka *PrintPage*. Tiskanje izvedemo s pomočjo metode *DrawImage* razreda *Graphics*. Metoda ima kar 30 različnih preobtežitev, v naslednjem primeru pa je prikazana ena od njih.

```
//printDocument3 -> tiskanje slike
private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
    float leviRob = e.MarginBounds.Left; //oddaljenost od levega roba
    float zgornjiRob = e.MarginBounds.Top;//oddaljenost od vrha strani
    //sliko natisnemo z eno od metod DrawImage - ta ima 30 preobtežitev
    e.Graphics.DrawImage(slika,leviRob,zgornjiRob);
}
```

Menjalnica

Radi bi odprli svojo menjalnico in zato želimo napisati ustrezen program. Obrazec pripravimo tak, da bo uporabnik lahko izbral valuto, ki jo ima in valuto v katero želi pretvoriti svoj znesek. Za izbiro potrebujemo dva gradnika tipa *ComboBox*. Znesek za prevorbo bo vnesel v gradnik tipa *TextBox*.



Slika 31: Seznam gradnikov projektu *Menjalnica*.

Ažurne podatke o valutah in valutnih tečajih bomo uvozili s spletnega naslova *Nove Ljubljanske Banke* http://www.nlb.si/?a=tecajnica&type=companies&format=xml. Podatki so v *XML* obliki, preko objekta tipa *DataSet* pa jih bomo uvozili v gradnik *DataGridView* in še v dva gradnika *ComboBox*.

```
public partial class Form1 : Form
{
```

```
    System.Data.DataSet ds; //dinamična tabela za uvoz podatkov
    public Form1()
    {
        InitializeComponent();
        try
        {
            lNapis.Text = "V eno izmed spodnjih polj vnesite znesek za
preračun\n\rin izberite valuti.";
            //velikost obrazca
            this.Height = 645;
            this.Width = 360;
            //Uvoz podatkov v DataGridView
            ds = new System.Data.DataSet();
            //podatek uvozimo z metodo ReadXml objekta ds
ds.ReadXml(@"http://www.nlb.si/?a=tecajnica&type=companies&format=xml");
            this.dataGridView1.DataSource = ds;
            //določim ime tabele iz podatkovnega izvora
            dataGridView1.DataMember = "Rate";
            //gradnik DataGridView primerno oblikujemo
            dataGridView1.DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleCenter;
            dataGridView1.Columns[0].HeaderText = "Valuta";
            dataGridView1.Columns[0].Width = 50;
            dataGridView1.Columns[1].HeaderText = "Tečaj";
            dataGridView1.Columns[1].Width = 60;
            dataGridView1.Columns[2].HeaderText = "Enota";
            dataGridView1.Columns[2].Width = 40;
            dataGridView1.Columns[2].DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleCenter;
            dataGridView1.Columns[3].HeaderText = "Nakup";
            dataGridView1.Columns[3].DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
            dataGridView1.Columns[4].HeaderText = "Prodaja";
            dataGridView1.Columns[4].DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;
            dataGridView1.RowHeadersVisible = false;
            //v oba spustna seznama dodamo najprej domačo valuto
            cBImam.Items.Add("EUR");
            cBDobim.Items.Add("EUR");
            //v oba spustna seznama dodamo še valute iz tečajnice
            for (int i = 0; i < dataGridView1.Rows.Count; i++)
            {
        cBImam.Items.Add(dataGridView1.Rows[i].Cells[0].Value.ToString());
        cBDobim.Items.Add(dataGridView1.Rows[i].Cells[0].Value.ToString());
            }
            cBImam.Sorted = true;//podatke uredimo po abecedi
            cBImam.Text = "EUR"; //privzeta valuta
            cBDobim.Sorted = true;
            cBDobim.Text = "EUR";
            /*uredimo še podatke v gradniku DataGridView - uredimo jih po
```

```
                prvem stolpcu*/
            dataGridView1.Sort(dataGridView1.Columns[0],
ListSortDirection.Ascending);
        }
        catch
        { MessageBox.Show("Tečajna lista trenutno ni dosegljiva!"); }

    }
}
```

Na obrazcu bo možen vnos zneska, ki ga želimo zamenjati v poljubno valuto. Možen bo tudi obraten izračun: kolikšen znesek v poljubni valuti potrebujemo, da dobimo želeni znesek v neki drugi valuti. V ta namen sta na obrazcu dva gradnika *TextBox* (*tBImam* in *tBDobim*). Obema bomo priredili tudi odzivno metodo dogodka *KeyPress*, da bo možen le vnos števk in decimalne vejice. Ko eden od njiju postane aktiven (klik z miško), se mu barva ozadja spremeni, obenem pa se spremenita napisa v gradnikih *GroupBox*, ki označujeta smer preračunavanja (*Imam* → *Potrebujem* oz. *Dobim* → *Želim*). Sprememba vsebine enega od gradnikov *TextBox* povzroči takojšno spremembo drugega gradnika, razen v primeru, da sta izbrani valuti enaki. V programu to dosežemo s pomočjo odzivnih metod dogodkov *TextChanged*, ki pa ju gradnikom *tBImam* in *tBDobim* ne moremo prirediti že v fazi razvoja projekta (v oknu *Properties*). Sprememba enega gradnika bi namreč povzročila spremembo drugega, kar bi se ponavljalo v neskončnost in program bi se sesul. Odzivni metodi moramo zato obema gradnikoma prirejati dinamično. Ko postane aktiven gradnik *tBImam* v seznam metod, ki obdelujejo dogodke, dodamo odzivno metodo dogodka *TextChanged* gradnika *tBImam*, s seznama pa odstranimo odzivno metodo dogodka *TextChanged* gradnika *tBDobim*. Ob kliku v polje *tBDobim* pa storimo ravno obratno.

```
//odzivna metoda dogodka KeyPress gradnikov tBImam in tBDobim
private void tBImam_KeyPress(object sender, KeyPressEventArgs e)
{
    //dovoljen je le vnos cifer, dovoljeno je tudi brisanje znakov
    if ((e.KeyChar < '0' || e.KeyChar > '9') && e.KeyChar != (char)8)
        e.Handled = true;
}
private void tBImam_Enter(object sender, EventArgs e)
{
    /*ko se z miško postavimo v polje se mu barva spremeni, spremeni se
      tudi napis gradnika GroupBox*/
    tBImam.BackColor = Color.Gold;
    tBDobim.BackColor = Color.White;
    gBImam.Text = "Imam";
    gBDobim.Text = "Dobim";
    //v seznam metod, ki obdelujejo dogodke, dodamo nov dogodek
    tBImam.TextChanged += new EventHandler(tBImam_TextChanged);
    //iz seznama metod odstranimo dogodek
    tBDobim.TextChanged -= new EventHandler(tBDobim_TextChanged);
}
//odzivna metoda dogodka Enter ob vstopu v gradnik tBImam
private void tBDobim_Enter(object sender, EventArgs e)
```

```
{
    /*ko se z miško postavimo v polje se mu barva spremeni, spremeni se
      tudi napis gradnika GroupBox*/
    tBImam.BackColor = Color.White;
    tBDobim.BackColor = Color.Gold;
    gBImam.Text = "Potrebujem";
    gBDobim.Text = "Želim";
    //v seznam metod, ki obdelujejo dogodke, dodamo nov dogodek
    tBDobim.TextChanged += new EventHandler(tBDobim_TextChanged);
    //iz seznama metod odstranimo dogodek
    tBImam.TextChanged -= new EventHandler(tBImam_TextChanged);
}
//odzivna metoda dogodka TextChanged ob spremembi vsebine gradnika TBImam
private void tBImam_TextChanged(object sender, EventArgs e)
{
    Izracunaj(1);//izračun, če je aktivno polje tBImam
}
//odzivna metoda dogodka TextChanged ob spremembi vsebine gradnika TBDobim
private void tBDobim_TextChanged(object sender, EventArgs e)
{
    Izracunaj(2);//izračun, če je aktivno polje tBDobim
}

double tecaj1, tecaj2;
//lastna metoda za izračun zneska pretvorbe
private void Izracunaj(int N)
{
    if (cBImam.Text == cBDobim.Text)
    {
        if (N == 1)//če aktivno polje tBImam
            tBDobim.Clear();
        else       //če aktivno polje tDobim
            tBImam.Clear();
    }
    else
    {
        //če znesek v tBImam  --->>> upoštevamo prodajne tečaje!!!
        //če znesek v tBDobim --->>> upoštevamo nakupni tečaj
        double znesek, rezultat;
        tecaj1 = 1; tecaj2 = 1;
        try
        {
            if (N == 1) //če aktivno polje tBImam
                znesek = Convert.ToDouble(tBImam.Text);
            else //če aktivno polje tBDobim
                znesek = Convert.ToDouble(tBDobim.Text);
            if (cBImam.Text == "EUR") //pretvorba iz EUR v neko valuto
            {
                //poiščemo ustrezen tečaj glede na izbrano valuto
                for (int i = 0; i < dataGridView1.Rows.Count; i++)
```

```
                {
                    if (dataGridView1.Rows[i].Cells[0].Value.ToString() ==
cBDobim.Text)
                    {
                        tecaj2 =
Convert.ToDouble(dataGridView1.Rows[i].Cells[4].Value);
                        break;
                    }
                }
            }
            else if (cBDobim.Text == "EUR") //pretvorba EUR v neko valuto
            {
                //poiščemo ustrezen tečaj glede na izbrano valuto
                for (int i = 0; i < dataGridView1.Rows.Count; i++)
                {
                    if (dataGridView1.Rows[i].Cells[0].Value.ToString() ==
cBImam.Text)
                    {
                        tecaj1 =
Convert.ToDouble(dataGridView1.Rows[i].Cells[3].Value);
                        break;
                    }
                }
            }
            else  //pretvorba iz valute v valuto
            {
                for (int i = 0; i < dataGridView1.Rows.Count; i++)
                {
                    //poiščemo ustrezna tečaja glede na izbrano valuto
                    if (dataGridView1.Rows[i].Cells[0].Value.ToString() ==
cBImam.Text)
                        tecaj1 =
Convert.ToDouble(dataGridView1.Rows[i].Cells[3].Value);
                    if (dataGridView1.Rows[i].Cells[0].Value.ToString() ==
cBDobim.Text)
                        tecaj2 =
Convert.ToDouble(dataGridView1.Rows[i].Cells[4].Value);
                }
            }
            if (N == 1)//če aktivno polje tBImam
                rezultat = znesek / tecaj1 * tecaj2;
            else //če aktivno polje tBDobim
                rezultat = znesek / tecaj2 * tecaj1;
            if (N == 1) //če aktivno polje tBImam
                tBDobim.Text=String.Format("{0:###,##0.00;(-
##,##0.00);Nič}",rezultat);
            else //če aktivno polje tBDobim
                tBImam.Text = String.Format("{0:###,##0.00;(-
##,##0.00);Nič}", rezultat);
        }
```

```
        catch
        {
            /*če je pri izračunu prišlo do napake pobrišemo vsebino
              ustreznega TextBox-a*/
            if (N == 1) //če aktivno polje tBImam
                tBDobim.Clear();
            else tBImam.Clear();//če aktivno polje tBDobim
        }
    }
}
//odzivna metoda dogodka DropDownClosed gradnikov tipa ComboBox
private void cBImam_DropDownClosed(object sender, EventArgs e)
{
    if (gBImam.Text == "Imam")
        Izracunaj(1);
    else Izracunaj(2);
}
```

Za preračun zneska v izbrano valuto smo napisali lastno metodo *Izracunaj*. Kot parameter smo ji poslali oznako, preko katere ugotovimo, v katero polje je uporabik vnesel znesek.

Za opravljeno menjavo želimo pripraviti tudi izpis na tiskalniku. Pred tiskanjem najprej preverimo, če bila menjava že izvedena. Izpis bi si radi ogledali najprej v predogledu.

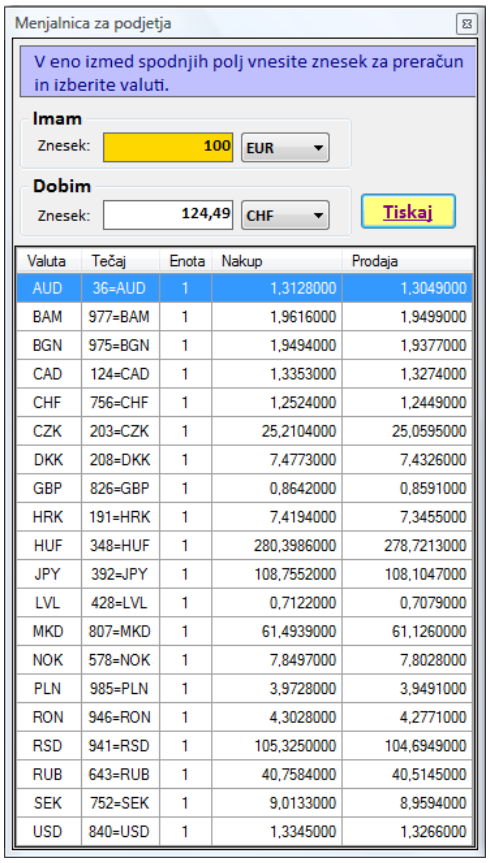
```
Image logo; //najava objekta za sliko
//odzivna metoda dogodka Klik gumba Tiskaj
private void bTiskaj_Click(object sender, EventArgs e)
{
    if (tBImam.Text.Trim() == "" || tBDobim.Text.Trim() == "")
    {
        MessageBox.Show("Najprej vnesi ustrezen znesek!","Izpis?",0 ,
                        MessageBoxIcon.Information);
    }
    else
    {
        printDocument1.DocumentName = "Menjalnica!";
        //slika oz. logotip, ki ga bomo natisnili
        logo = new Bitmap("Menjalnica.jpg"    , true);
        printPreviewDialog1.Document = printDocument1;
        /*Pred dokončnim tiskanjem MORAMO definirati še odzivno metodo
          dogodka PrintPage, kjer povemo, kaj in kako bomo tiskali*/
        printPreviewDialog1.ShowDialog();
    }
}
```

Na vrhu izpisa naj bo v desnem zgornjem kotu tekoči datum, v levem kotu logotip našega podjetja, ob njem pa naš naziv. Sledili bodo podatki o sami menjavi, ter menjalniškem tečaju. Pred tiskanjem zneskov bomo le-te ustrezno formatirali, za tiskanje pa uporabili font, v katerem

so znaki stalno enake širine, npr.*Courier New*. Kot smo že zapisali, izpis opremimo s sliko in nekaj vodoravnimi črtami.

```
Font printFont; //najava objekta za določitev izbranega fonta
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    float x = e.MarginBounds.Left; //oddaljenost od levega roba
    float zgornjiRob = e.MarginBounds.Top;//oddaljenost od vrha strani
    float sirina = e.MarginBounds.Width; //širina izpisa
    printFont = new Font("Calibri", 10, FontStyle.Bold | FontStyle.Italic);
    float y = zgornjiRob; //oddaljenost izpisa od zgornjega roba
    string datum = DateTime.Now.ToLongDateString();
    //izpis kraja in datuma v desnem zgornjem kotu
    e.Graphics.DrawString("Kranj, "+datum, printFont, Brushes.Gray, x + 470,
y, new StringFormat());
    e.Graphics.DrawImage(logo, x, zgornjiRob+20);    //izpis logotipa
    printFont = new Font("Calibri", 20, FontStyle.Bold | FontStyle.Italic);
    y = y + 50;
    //izpis naziva in naslova menjalnice
    e.Graphics.DrawString("Menjalnica EURO ", printFont, Brushes.Gray, x+200,
y, new StringFormat());
    printFont = new Font("Calibri", 14, FontStyle.Bold | FontStyle.Italic);
    y = y + 2*printFont.GetHeight(e.Graphics);
    e.Graphics.DrawString("Trubarjeva 12", printFont, Brushes.Gray, x + 200,
y, new StringFormat());
    y = y + printFont.GetHeight(e.Graphics);
    e.Graphics.DrawString("4000    KRANJ ", printFont, Brushes.Gray, x + 200,
y, new StringFormat());
    y = y + 70;
    //vodoravna črta debeline 2
    e.Graphics.DrawLine(new Pen(Color.Gray, 2),x, y, x + sirina, y);
    printFont = new Font("Calibri", 12, FontStyle.Bold | FontStyle.Regular);
    y = y + 15;
    //glava izpisa
    e.Graphics.DrawString("Valuta za prodajo                          Znesek
Nova valuta            Znesek v novi valuti", printFont, Brushes.Gray, x
+10, y, new StringFormat());
    y = y + 20;
    //vodoravna črta debeline 1
    e.Graphics.DrawLine(new Pen(Color.Gray, 1), x, y, x + sirina, y);
    double znesek = Convert.ToDouble(tBImam.Text);
    double ZnesekvValuti = Convert.ToDouble(tBDobim.Text);
    double noviZnesek = Convert.ToDouble(tBDobim.Text);
    //oba zneska formatiramo
    string sZnesek = String.Format("{0:###,##0.00}", znesek);
    string sNoviZnesek = String.Format("{0:###,##0.00}", noviZnesek);
    //za izpis zneskov izberemo font stalne širine
```

```
    printFont = new Font("Courier new", 10, FontStyle.Bold |
FontStyle.Regular);
    //niz za izpis ustrezno formatiramo
    string izpis =
string.Format("{0,10}{1,21}{2,10}{3,25}",cBImam.Text,sZnesek,cBDobim.Text,sNo
viZnesek);
    y = y + 20;
    e.Graphics.DrawString(izpis, printFont, Brushes.Gray, x + 10, y, new
StringFormat());
    y = y + 15;
    e.Graphics.DrawLine(new Pen(Color.Gray, 1), x, y, x + sirina, y);//črta
    y = y + 40;
    //izpišemo še podatek o uporabljenem prodajnem tečaju
    izpis = "Prodajni tečaj: " + tecaj2.ToString();
    e.Graphics.DrawString(izpis, printFont, Brushes.Gray, x + 10, y, new
StringFormat());
    printFont = new Font("Calibri", 12, FontStyle.Bold | FontStyle.Regular);
    y = y + 15;
    //vodoravna črta
    e.Graphics.DrawLine(new Pen(Color.Gray, 1), x, y, x + sirina, y);
}
```



Valuta	Tečaj	Enota	Nakup	Prodaja
AUD	36=AUD	1	1,3128000	1,3049000
BAM	977=BAM	1	1,9616000	1,9499000
BGN	975=BGN	1	1,9494000	1,9377000
CAD	124=CAD	1	1,3353000	1,3274000
CHF	756=CHF	1	1,2524000	1,2449000
CZK	203=CZK	1	25,2104000	25,0595000
DKK	208=DKK	1	7,4773000	7,4326000
GBP	826=GBP	1	0,8642000	0,8591000
HRK	191=HRK	1	7,4194000	7,3455000
HUF	348=HUF	1	280,3986000	278,7213000
JPY	392=JPY	1	108,7552000	108,1047000
LVL	428=LVL	1	0,7122000	0,7079000
MKD	807=MKD	1	61,4939000	61,1260000
NOK	578=NOK	1	7,8497000	7,8028000
PLN	985=PLN	1	3,9728000	3,9491000
RON	946=RON	1	4,3028000	4,2771000
RSD	941=RSD	1	105,3250000	104,6949000
RUB	643=RUB	1	40,7584000	40,5145000
SEK	752=SEK	1	9,0133000	8,9594000
USD	840=USD	1	1,3345000	1,3266000

Slika 32: Projekt *Menjalnica* v uporabi.


Kranj 31. december 2010

Menjalnica EURO

Trubarjeva 12
4000 KRANJ

Valuta za prodajo	Znesek	Nova valuta	Znesek v novi valuti
EUR	100,00	CHF	124,49

Prodajni tečaj: 1,2449

Slika 33: Projekt *Menjalnica*: primer izpisa na tiskalniku.

Povzetek

Spoznali smo osnove objektnega programiranja v vizuelnem okolju – dedovanje in polimorfizem. Naučili so se izdelovanja klasičnih in *MDI* večokenskih aplikacij, prikazali smo tudi delo s tiskalnikom. Pri izdelavi izpisa si lahko pomagamo z gradniki na paleti *Printing*, a postopek je dokaj zahteven in zamuden. Veliko lepše izpise in poročila lahko izdelamo s pomočjo orodij, ki so na voljo na spletu (večinoma proti plačilu), npr. *Crystal Reports*.

V prikazanih primerih smo uporabili številne nove gradnike, predvsem z namenom, da prikažemo njihove zmožnosti in namen uporabe. Pri tem smo spoznali nove razrede in njihove lastnosti ter metode. Nemogoče si je zapomniti vse, pomembno pa je, da jih znamo uporabiti in nam je njihova uporaba v zapisanih primerih in vajah razumljiva. Pri reševanju novih projektov bo tako dovolj, da se spomnimo primera, kjer smo nekaj podobnega že reševali. Pogledali bomo takratno rešitev, ali pa si podobno rešitev pogledali v tej literaturi. Potrebno bo le še poiskati dodatne informacije in prodobljeno znanje uporabiti v novih situacijah. Odločilno vlogo pri tem pa predstavlja zmožnost analitičnega povezovanja že pridobljenega znanja.

Seznam krvodajalcev

Na začetku poglavja smo si zastavili pripravo večokenske aplikacije za vodenje seznama o krvodajalcih. Znanje, ki smo ga pridobili v tem poglavju nam sedaj to omogoča. Na glavnem obrazcu projekta bomo prikazali uporabo gradnika *ListView* s seznamom, zato projekt poimenujmo *ListViewKrvodajalci*. Gradnik *ListView* omogoča tabelarični prikaz podatkov, poleg tega pa lahko podatke tudi grupiramo.

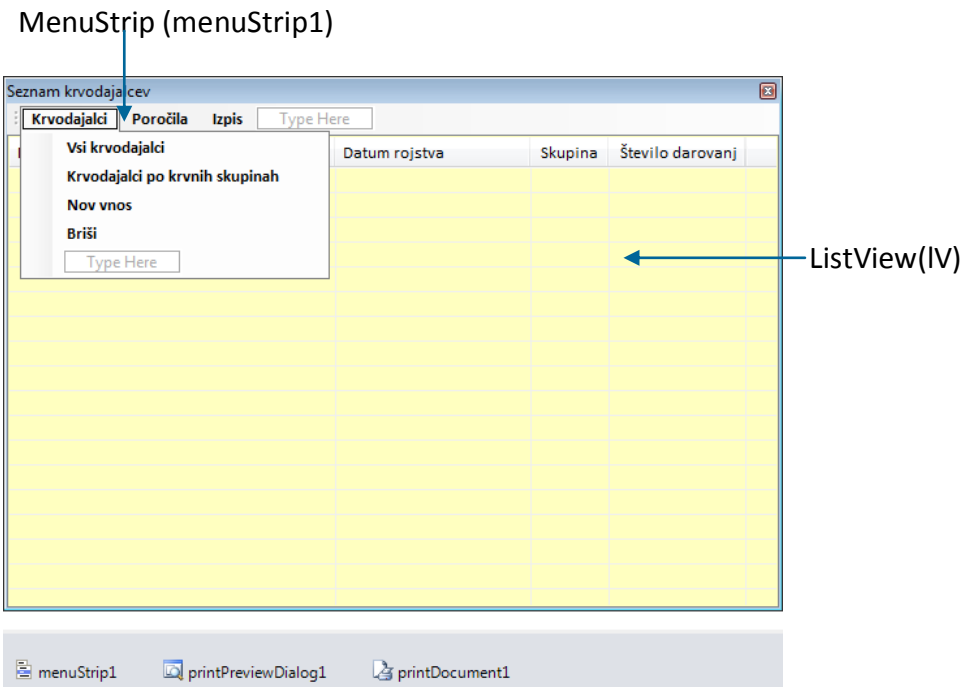
V projekt najprej vključimo nov razred. V *Solution Explorer*-ju z desnim klikom na projektno datoteko izberemo možnost *Add*, nato *New* Item in Class.*cs*. Ime datoteke spremenimo v *Razredi.cs* in vnos potrdimo s klikom na gumb *Add*. V datoteko zapišemo osnovni razred *Oseba* in iz njega izpeljimo razred *Krvodajalec*.

```
class Oseba //osnovni razred
{
    protected string naziv;
    protected DateTime datumRojstva;
    public Oseba()//privzeti konstruktor
    {}
    //preobteženi konstruktor
    public Oseba(string naziv,DateTime datum)
    {
        this.naziv=naziv;
        datumRojstva=datum;
    }
}

//izpeljani razred
class krvodajalec : Oseba
{
    private string krvnaSkupina;
    private int steviloDarovanj;
    public krvodajalec()//privzeti konstruktor deduje osnovnega
        :base()
    { }

    //preobteženi konstruktor prav tako deduje osnovni konstruktor
    public krvodajalec(string naziv, DateTime datum, string krvnaSkupina,int
steviloDarovanj)
        : base(naziv, datum)
    {
```

```
        this.krvnaSkupina = krvnaSkupina;
        this.steviloDarovanj = steviloDarovanj;
    }
    public int SteviloDarovanj
    {
        get { return steviloDarovanj; }
        set { steviloDarovanj = value;}
    }
}
```

Zaradi enostavnosti v konstruktorjih obeh razredov in lastnosti izpeljnaga razreda nismo preverjali smiselnost prirejanja.

Glavni obrazec projekta poimenujmo *FSeznam*. V glavni meni na obrazcu zapišimo tri postavke (*Krvodajalci*, *Poročila* in *Izpis*). Prvi dve možnosti nato še razvejimo: postavka *Krvodajalci* naj ima štiri dodatne možnosti (glej sliko), postavka *Poročila* pa dve (*Število krvodajalcev po krvnih skupinah* in *Najboljši krvodajalci*).

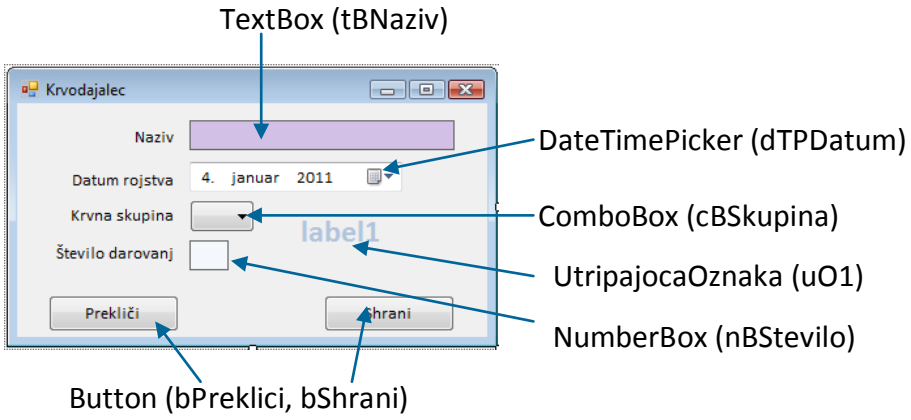


Slika 34: Glavni obrazec projekta *Seznam krvodajalcev*.

Gradnik tipa *ListView*, smo poimenovali *lV,* mu nastavili lastnost *Dock* na *Fill*, *Sorting* pa na *Ascending*. Seznam bomo zato lahko urejali po abecedi prvega stolpca. *GridLines* smo nastavili na *True*, da so med vrsticami prikazane vodoravne črte. Lastnost *View* je nastavljena na podrobnostni pregled *Details*. Stolpce smo ustvarili na podoben način kot smo jih pri gradniku tipa *DataGridView*. V oknu *Properties* kliknimo *Columns*. V oknu *ColumnHeather Collection Editor* nato s klikom na gumb *Add* dodajmo štiri stolpce in jih s pomočjo lastnosti (*Name*)

poimenujmo *Naziv*, *Datum*, *Skupina* in *SteviloDarovanj*. Določili smi jim še lastnost Text, to je napis na vrhu stolpcev: *Naziv*, *Datum rojstva*, *Skupina* in *Število darovanj*. Ostale lastnosti stolpcev smo pustili privzete. Imena in število grup, s pomočjo katerih bomo krvodajalce lahko grupirali, bomo za vajo določili programsko. Grupe bi seveda lahko določili tudi s klikom na lastnost *Groups* v oknu *Properties* in imena grup ustvarili na podoben način kot stolpce. Za potrebe izpisa in predogleda izpisa potrebujemo še gradnika *PrintPreviewDialog* in *PrintDocument*. Ob zagonu projekta bomo podatke v gradnik tipa *ListView* uvozili iz tekstovne datoteke *Krvodajalci.txt*, ki se nahaja v mapi *Bin→Debug* našega projekta.

Pripraviti moramo še obrazec za vnos oz. ažuriranje krvodajalca. V projekt dodamo nov obrazec (*Project→Add Windows Form…*) in ga poimenujmo *Krvodajalec*. Gradniku *cBSkupina* preko lastnosti *Items* dodamo ustrezne krvne skupine. Poskrbimo tudi za odzivno metodo modalnega gumba z naoisom *Shrani*. Obrazec se ne bo zaprl, če uporabnik ne vnese naziva in števila darovanj. Na obrazcu sta še gradnika tipa *UtripajocaOznaka* in *NumberBox*. To sta vizuelna gradnika, ki smo ju ustvarili v knjižnici *MojaKnjiznica* v prejšnjem poglavju. Če hočemo imeti dostop do obeh, moramo v projekt uvoziti omenjeno knjižnico. To že znamo: v oknu *SolutionExplorer* izberemo ime projekta *ListViewKrvodajalci → desni klik → AddReference → Browse* in poiščemo ter izberemo knjižnico *MojaKnjiznica.dll*.



Slika 35: Obrazec za Vnos/Ažuriranje podatkov krvodajalca.

S stavkom

```
using MojaKnjiznica;
```

knjižnico dodamo v *using* sekcijo datoteke *Krvodajalec.cs*. Vsem gradnikom na obrazcu določimo lastnost *Modifiers* na *Public*. Do njih bomo zato lahko dostopali tudi iz drugih obrazcev. Gumb *bPreklici* ima lastnost *DialogResult* nastavljeno *Cancel*, gumb *bShrani* pa OK!

Lotimo se odzivnih metod glavnega obrazca. V konstruktorju najprej poskrimo za programsko dodajanje imen in opisa štrih grup objekta *IV*. To so osnovne štiri krvne skupine.

Obstoječe podatke o krvodajalcih imamo že zapisane v datoteki *Krvodajalci.txt*. Datoteko zato odpremo za branje. Beremo podatke za vsakega krvodajalca posebej in zanj ustvarimo nov objekt tipa *ListViewItem*, ki predstavlja vrstico gradnika *ListView*. Z metodo *Add* nato vrstico dodamo v obstoječi seznam.

```
public partial class FSeznam : Form
{
    public FSeznam() //konstruktor glavnega obrazca
    {
        InitializeComponent();
        //Izdelajmo grupe
        lV.Groups.Add("A", "KRVNA SKUPINA A");
        lV.Groups.Add("B", "KRVNA SKUPINA B");
        lV.Groups.Add("AB", "KRVNA SKUPINA AB");
        lV.Groups.Add("0", "KRVNA SKUPINA 0");

        //Podatke uvozimo iz datoteke Krvodajalci.txt
        StreamReader beri = File.OpenText("Krvodajalci.txt");
        string stavek = beri.ReadLine();//beremo prvo vrstico

        while (stavek != null)
        {
            string[] postavke = stavek.Split(';');
            int stGrupe=0;
            //določimo indeks grupe
            for (int i = 0; i < lV.Groups.Count; i++)
            {
                if (postavke[2] == lV.Groups[i].Name)
                {
                    stGrupe = i;
                    break;
                }
            }
            /*Ustvarimo objekt tipa ListViewItem – nova vrstica gradnika
             ListView: konstruktor potrebuje za parameter tabelo, ki hrani
             vsebino stolpcev in grupo v katero spada ta vrstica*/
            ListViewItem vstavi = new ListViewItem(new string[] {
postavke[0], postavke[1], postavke[2], postavke[3] }, lV.Groups[stGrupe]);
            vstavi.Group.Name = postavke[2];
            //Novo postavko dodamo v gradnik ListView z imenom lV
            lV.Items.Add(vstavi);

            stavek = beri.ReadLine();//beremo naslednjo vrstico
        }
        beri.Close();
    }
}
```

Tabelo krvodajalcev imamo lahko prikazano na dva načina: pogleda po grupah in običajen pogled. V ta namen napišemo prvi dve odzivni metodi postavke Krvodajalci glavnega menija.

```
//odzivna metoda za klasični pogled na podatke – brez prikaza grup
private void vsiKrvodajalciToolStripMenuItem_Click(object sender,EventArgs e)
{
    lV.ShowGroups = false;
}
//odzivna metoda za prikaz seznama po grupah
private void krvodajalciPoKrvnihSkupinahToolStripMenuItem_Click(object
sender, EventArgs e)
{
    lV.ShowGroups = true;
}
```

Novega krvodajalca bomo vnesli preko prej pripravljenega obrazca *Krvodajalec.cs*. Obrazec odpremo modalno, vnesene podatke pa nato preko objekta razreda *ListViewItem* dodamo v seznam.

```
/*nov vnos realiziramo s pomočjo obrazca z imenom Krvodajalec – obrazec
  moramo pripraviti posebej*/
private void novVnosToolStripMenuItem_Click(object sender, EventArgs e)
{
    lV.ShowGroups = true;
    Krvodajalec Nov = new Krvodajalec();
    //poskrbimo za začetno izbiro krvne skupine
    Nov.cBSkupina.SelectedIndex = 0;
    Nov.uO1.label1.Text = "Vnos";//na objektu označimo, da gre za vnos
    if (Nov.ShowDialog() == DialogResult.OK)//uporabnik je kliknil OK
    {
        /*V ListView lahko dodamo celo vrstico naenkrat: prvi parameter
          za konstruktor je tabela z vsebino celic, drugi parameter pa
          indeks, s katerim povemo, kateri grupi vrstica pripada*/
        ListViewItem vstavi = new ListViewItem(new string[]
{Nov.tBNaziv.Text, Nov.dTPDatum.Value.ToLongDateString(),
Nov.cBSkupina.Text,Nov.nBStevilo.Text
},lV.Groups[Nov.cBSkupina.SelectedIndex]);
        vstavi.Group.Name = Nov.cBSkupina.Text; //določimo še ime grupe
        lV.Items.Add(vstavi); //vrstico dopišemo v seznam lV
    }
}
```

Dodajmo še možnost brisanja krvodajalca iz seznama. Napišimo odzivno metodo možnosti *Briši* postavke *Krvodajalci* glavnega menija. V njej najprej ugotovimo indeks aktivne vrstice, ki jo nato odstranimo iz seznama. Uporabimo tudi varovalni blok.

```
//metoda za brisanje izbrane vrstice
private void brišiToolStripMenuItem_Click(object sender, EventArgs e)
{
```

```
    try
    {
        int vrstica = lV.SelectedIndices[0];
        string naziv = lV.Items[vrstica].Text;
        if (MessageBox.Show(naziv+":
Brišem?","BRISANJE",MessageBoxButtons.YesNo,MessageBoxIcon.Question)==DialogR
esult.Yes)
            lV.SelectedItems[0].Remove();
    }
    catch { }
}
```

Podatke o krvodajalcih v seznamu želimo tudi urejati. Zato bo skrbela odzivna metoda dogodka *DoubleClick* objekta *lV*. Uporabimo kar že pripravljeni obrazec Krvodajalec.cs iz katerga ustvarimo nov objekt z imenom *Nov*. Preden ga prikažemo, nanj prenesemo podatke o izbranem krvodajalcu. Do podatkov v prvem stolpcu izbrane vrstice dostopamo s pomočjo lastnosti *Items* in indeksa vrstice. Za ostale stolpce pa moramo dodati še lastnost *SubItems* in indeks stoplca.

```
//dvoklik na gradnik lV je namenjen urejanju podatkov izbrane vrstice
private void listView1_DoubleClick(object sender, EventArgs e)
{
    int vrstica = lV.SelectedIndices[0]; //indeks izbrane vrstice
    //podatke izbrane vrstice zapišemo na obrazec Krvodajalec
    Krvodajalec Nov = new Krvodajalec();
    Nov.tBNaziv.Text = lV.Items[vrstica].Text;
    Nov.dTPDatum.Value =
Convert.ToDateTime(lV.Items[vrstica].SubItems[1].Text);
    Nov.cBSkupina.Text = lV.Items[vrstica].SubItems[2].Text;
    Nov.nBStevilo.Text = lV.Items[vrstica].SubItems[3].Text;
    Nov.uO1.label1.Text = "Ažuriranje";
    if (Nov.ShowDialog() == DialogResult.OK)//uporabnik kliknil gumb OK
    {
        /*če je uporabnik kliknil gumb Shrani, se podatki zapišejo
          nazaj v isto vrstico gradnika lV*/
        lV.Items[vrstica].Text = Nov.tBNaziv.Text;
        /*LAHKO TUDI: listView1.SelectedItems[0].Text =
          Nov.textBox1.Text;*/
        lV.Items[vrstica].SubItems[1].Text =
Convert.ToString(Nov.dTPDatum.Value.ToLongDateString());
        lV.Items[vrstica].SubItems[2].Text = Nov.cBSkupina.Text;
        lV.Items[vrstica].SubItems[3].Text = Nov.nBStevilo.Text;
        lV.Items[vrstica].Group = lV.Groups[Nov.cBSkupina.Text];
    }
}
```

Za ugotavljanje števila krvodajlec po krvnih skupinah in ugotavljanje najboljših krvodajalcev napišimo odzivni metodi postavke *Poročila* glavnega menija.

```
//ugotavljanje števila krvodajalcev po krvnih skupinah
private void obdelavaToolStripMenuItem_Click(object sender, EventArgs e)
{
    int[] tabSkupin = new int[4];
    for (int i=0;i<lV.Items.Count;i++)
        switch (lV.Items[i].SubItems[2].Text)
        {
            case "A": tabSkupin[0]++; break;
            case "B": tabSkupin[1]++; break;
            case "AB": tabSkupin[2]++; break;
            case "0": tabSkupin[3]++; break;
        }
    MessageBox.Show("Krvna skupina A: "+tabSkupin[0].ToString()+
                        "\nKrvna skupina B: "+tabSkupin[1].ToString()+
                        "\nKrvna skupina AB: "+tabSkupin[2].ToString()+
                        "\nKrvna skupina 0: " + tabSkupin[3].ToString()+
                        "\n-------------------------"+
                        "\nSkupaj: "+lV.Items.Count.ToString()+
                        "\n================");
}

//ugotavljanje najboljših krvodajalcev
private void največjiKrvodajalecToolStripMenuItem_Click(object sender,
EventArgs e)
{
    string najv = "";
    int naj = 0;
    for (int i = 0; i < lV.Items.Count; i++)
    {
        if (Convert.ToInt32(lV.Items[i].SubItems[3].Text) > naj)
        {
            najv = lV.Items[i].SubItems[0].Text;
            naj = Convert.ToInt32(lV.Items[i].SubItems[3].Text);
        }
    }
    //še izpis zmagovalcev;
    string zmagovalci = "";
    for (int i = 0; i < lV.Items.Count; i++)
    {
        if (Convert.ToInt32(lV.Items[i].SubItems[3].Text) == naj)
                zmagovalci += lV.Items[i].SubItems[0].Text+"\n\r";
    }
    zmagovalci+="Število darovanj: "+naj.ToString();
    MessageBox.Show(zmagovalci, "\n\rNajboljši darovalec/ci", 0,
MessageBoxIcon.Information);
}
```

Ostane nam še izpis seznama na tiskalniku. Naredili bomo tudi predogled, v odzivni metodi dogodka PrintPage pa poskrbimo, da bo izpis tudi primerno oblikovan.

```
//priprava za izpis na tiskalniku - predogled
Font printFont; //pred tiskanjem bomo vsakič določili pisavo
string strPrint;//niz, v katerega bomo pisali besedilo za tiskanje
private void izpisToolStripMenuItem_Click(object sender, EventArgs e)
{
    printDocument1.DocumentName = "Seznam krvodajalcev";
    printPreviewDialog1.Document = printDocument1;
    printPreviewDialog1.ShowDialog();
}

//najpomembnejši del priprave za tiskanje pa je dogodek PrintPage
int zaporedna;//zaporedna številka izpisane vrstice
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    zaporedna = 1;//zaporedna številka izpisane vrstice
    float zgornjiRob = e.MarginBounds.Top;//oddaljenost od vrha strani
    float sirina = e.MarginBounds.Width; //širina izpisa
    float x = e.MarginBounds.Left; //oddaljenost od levega roba
    float y = zgornjiRob; //oddaljenost izpisa od zgornjega roba
    //glava strani
    printFont = new Font("Calibri", 14, FontStyle.Bold | FontStyle.Italic);
        e.Graphics.DrawString("SEZNAM KRVODAJALCEV", printFont,
Brushes.Black, x, y, new StringFormat());
    //oblikovanje glave stolpcev
    printFont = new Font("Calibri", 10, FontStyle.Regular |
FontStyle.Regular);
    y = y + 50;
    e.Graphics.DrawString("Naziv", printFont, Brushes.Black, x, y, new
StringFormat());
    e.Graphics.DrawString("Datum rojstva", printFont, Brushes.Black, x+280,
y, new StringFormat());
    e.Graphics.DrawString("Krvna skupina", printFont, Brushes.Black, x + 430,
y, new StringFormat());
    e.Graphics.DrawString("Število darovanj", printFont, Brushes.Black, x +
530, y, new StringFormat());
    //izpis vodoravne črte
    y = y + 20;
    e.Graphics.DrawLine(new Pen(Color.Gray, 2), x, y, x + sirina, y);
    int vrstica = 0; //maksimalno 45 postavk na stran
    //še izpis podatkov posameznega krvodajalca
    while (vrstica < 45 && zaporedna < lV.Items.Count - 1)
    {
        y = y + 20;
        e.Graphics.DrawString(lV.Items[zaporedna].Text,printFont,
Brushes.Black, x, y, new StringFormat());

e.Graphics.DrawString(Convert.ToDateTime(lV.Items[zaporedna].SubItems[1].Text
).ToLongDateString(), printFont, Brushes.Black, x + 280, y, new
StringFormat());
```

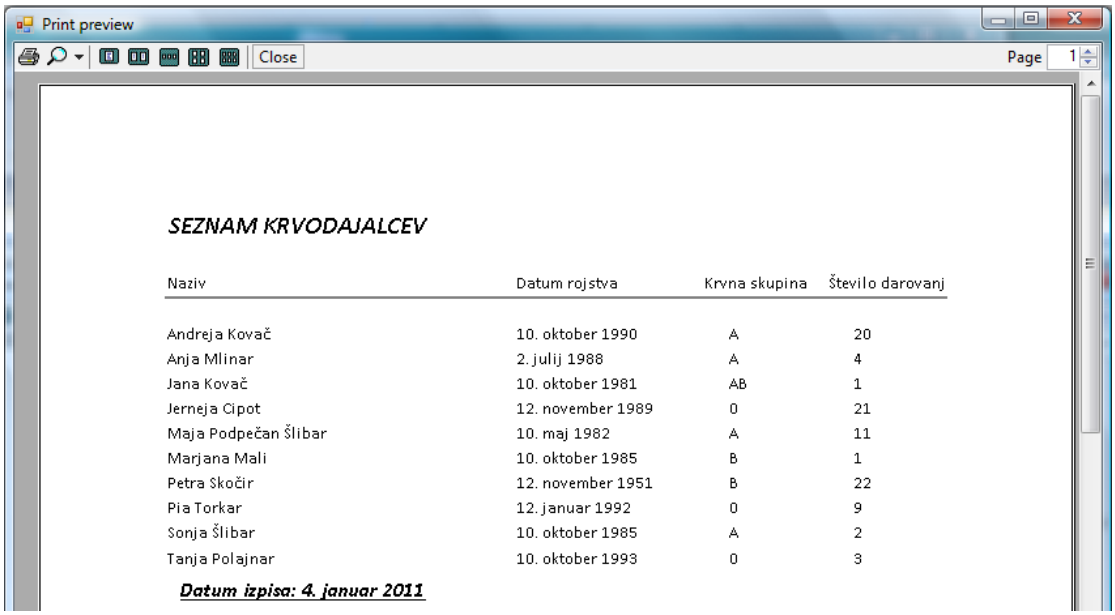
```
        e.Graphics.DrawString(lV.Items[zaporedna].SubItems[2].Text,
printFont, Brushes.Black, x + 450, y, new StringFormat());
        e.Graphics.DrawString(lV.Items[zaporedna].SubItems[3].Text,
printFont, Brushes.Black, x + 550, y, new StringFormat());
        vrstica++;
        zaporedna++;
    }

    //na koncu izpišemo še datum izpisa
    if (zaporedna == lV.Items.Count - 1)
    {
        printFont = new Font("Calibri", 12, FontStyle.Bold |
FontStyle.Italic|FontStyle.Underline);
        e.Graphics.DrawString("Datum izpisa: " +
DateTime.Now.ToLongDateString(), printFont, Brushes.Black, x + 10, y + 25,
new StringFormat());
    }

    //Preverimo, če je še kaj za natisniti
    if (zaporedna < lV.Items.Count - 1)
    {
        e.HasMorePages = true;
    }
    else
        e.HasMorePages = false;
    }
}
```

Pa še koda datoteke *Krvodajalec.cs*:

```
public partial class Krvodajalec : Form
{
    public Krvodajalec()
    {
        InitializeComponent();
    }

    /*ob kliku na gumb Shrani preverimo, ali je uporabnik vnesel ime in
     število darovanj*/
    private void bShrani_Click(object sender, EventArgs e)
    {
        if (tBNaziv.Text == "")
        {
            MessageBox.Show("Vnesi ime - ime ne sme biti prazno!");
            //Ker uporabnik ni vnesel imena se obrazec ne sme zapreti
            this.DialogResult = DialogResult.None;
        }
        else
            try
```

```
            {
                int stevilo = Convert.ToInt32(nBStevilo.Text);
            }
            catch
            {
                MessageBox.Show("Napaka pri vnosu števila
darovanj!\n\rŠtevilo darovanj mora biti celo število večje od 0!");
                /*Ker uporabnik pri vnosu števila naredil napako, se obrazec
                  ne sme zapreti*/
                this.DialogResult = DialogResult.None; ;
            }
    }
}
```

Tule je še primer končnega izpisa. Ta je seveda lahko večstranski, odvisno od števila krvodajalcev v seznamu.



Slika 36: Primer izpisa seznama krvodajalcev.

LITERATURA IN VIRI

- Uranič S. Praktično programiranje (online). Kranj: TŠC, 2010. (citirano 1. 2. 2011). Dostopno na naslovu: http://uranic.tsckr.si/C%23/Prakticno%20programiranje.pdf
- Sharp, J. *Microsoft Visual C# 2005 Step by Step*. Washington: Microsoft Press, 2005.
- Murach, J., in Murach, M. *Murach's C# 2008*. Mike Murach @ Associates Inc. London, 2008.
- Petric, D. *Spoznavanje osnov programskega jezika C#*, diplomska naloga. Ljubljana: UL FMF, 2008.
- Jerše, G., in Lokar, M. *Programiranje II*, *Objektno programiranje*. Ljubljana: B2, 2008.
- Jerše, G., in Lokar, M. *Programiranje II*, *Rekurzija in datoteke*. Ljubljana: B2, 2008.
- Kerčmar, N. *Prvi koraki v Javi, diplomska naloga*. Ljubljana: UL FMF, 2006.
- Lokar, M. *Osnove programiranja: programiranje – zakaj in vsaj kaj*. Ljubljana: Zavod za šolstvo, 2005.
- Uranič, S. *Microsoft C#.NET*, (online). Kranj: TŠC, 2008. (citirano 1. 2. 2011). Dostopno na naslovu: http://uranic.tsckr.si/C%23/C%23.pdf
- Uranič, S. *Microsoft Visual C#.NET*, (online). Kranj: TŠC, 2008. (citirano 1. 2. 2011). Dostopno na naslovu: http://uranic.tsckr.si/VISUAL%20C%23/VISUAL%20C%23.pdf
- *C# Practical Learning 3.0*, (online). 2008. (citirano 1.2.2011). Dostopno na naslovu: http://www.functionx.com/csharp/
- Lokar, M., et.al. *Projekt UP – Kako poučevati začetni tečaj programskega jezika, sklop interaktivnih gradiv*, (online). 2008. (citirano 1. 1. 2011). Dostopno na naslovu: http://up.fmf.uni-lj.si/
- Lokar, M. *Wiki C#*, (online). 2008 (citirano 1.2.2011). Dostopno na naslovu http://penelope.fmf.uni-lj.si/C_sharp
- Coelho, E. *Crystal Clear Icons*, (online). 2008 (citirano 1.2.2011). Dostopno na naslovu: http://commons.wikimedia.org/wiki/Crystal_Clear
- Mayo, J. *C# Station Tutorial*, (online). 2008 (citirano 1.2.2011). Dostopno na naslovu: http://www.csharp-station.com/Tutorial.aspx
- *C# Station*, (online). 2008 (citirano 1.2.2011). Dostopno na naslovu: http://www.csharp-station.com/Tutorial.aspx
- *CSharp Tutorial*, (online). 2008. (citirano 1.12.2011). Dostopno na naslovu: http://www.java2s.com/Tutorial/CSharp/CatalogCSharp.htm
- *FunctionX Inc*, (online). 2008. (citirano 1.2.20111). Dostopno na naslovu: http://www.functionx.com/csharp/
- *SharpDevelop, razvojno okolje za C#*, (online). 2008. (citirano 1. 2. 2011). Dostopno na naslovu: http: // www.icsharpcode.net/OpenSource/SD/.

- WIKI DIRI 06/07, (online). 2008. (citirano 1. 2. 2011). Dostopno na naslovu: http://penelope.fmf.uni-lj.si/diri0607/index.php/Kategorija:Naloge