Razvoj programskih aplikacij

RPA

Srečo Uranič

SPLOŠNE INFORMACIJE O GRADIVU

Izobraževalni program

Višja strokovna šola: Informatika, 2. letnik

Ime modula

Razvoj programskih aplikacij – RPA

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:
Lektor:
Datum:
CIP:

POVZETEK/PREDGOVOR

Gradivo ***Razvoj programskih aplikacij*** je namenjeno študsentom 2. letnika izobraževalnega programa VSŠ – Informatika. Pokriva vsebinski del modula Razvoj programskih aplikacij, kot je zapisan v katalogu znanja za ta program v višješolskem 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 Orirentirirano 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.

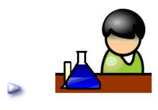
Literatura poleg teoretičnih osnov vsebuje številne primere in vaje. Vse vaje, pa tudi številne dodatne vaje in primeri so v stisnjeni obliki na voljo na strežniku ŠC Kranj `http://uranic.tsckr.si/VISUAL%20C%23/`, zaradi prevelikega obsega pa jih v literaturo nisem vključil še več.

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 ***Razvojprogramskih 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, osnove programiranja za splet, izdelava namestitvenega programa.

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

KAZALO

UČNI CILJI 12

CILJI 12

TRI V VRSTO 14

MICROSOFT Visual C# 2010 - OSNOVNI POJMI: Rešitve (Solutions) in projekti (projects) 14

Okenske aplikacije (Windows Forms) v Visual C# 2010 Express Edition 16

Ustvarjanje osnovnih okenskih aplikacij v okolju Visual C# 17

Lastnosti, metode in dogodki gradnikov, razred *object* 38

Sporočilno okno MessageBox 57

Delo z enostavnimi in sestavljenimi meniji, orodjarna 64

Glavni meni – MenuStrip (nadgradnja) 65

Lebdeči (Pop Up) meni - ContextMenuStrip 68

Orodjarna - ToolStrip 69

Gradnik StatusStrip 70

Dialogi – okenska pogovorna okna 70

ColorDialog – pogovorno okno za izbiro barve 71

FolderBrowserDialog – pogovorno okno za raziskovanje map 73

FontDialog – pogovorno okno za izbiro pisave. 76

OpenFileDialog - pogovorno okno za odpiranje datotek 77

SaveFileDialog - pogovorno okno za shranjevanje datotek 79

Gradnik DataGridView 89

Dogodki tipkovnice, miške in ure 101

Dogodki tipkovnice 101

Dogodki, ki jih sproži miška 108

Dogodki, ki jih sproži ura (gradnik *Timer*) in metode za delo s časom ____ 125

Dinamično ustvarjanje okenskih gradnikov in njihovih dogodkov ____ 135

Dinamično ustvarjanje okenskih gradnikov ____ 135

Dinamično ustvarjanje okenskih dogodkov ____ 136

Povzetek ____ 139

Tri v vrsto ____ 141

SEZNAM KRVODAJALCEV ____ 149

Dedovanje (inheritance) ____ 149

Osnovni razredi in izpeljani razredi ____ 149

Nove metode – operator new ____ 154

Virtualne in prekrivne metode ____ 154

Dedovanje vizuelnih gradnikov ____ 160

Izdelava lastne knjižnice razredov ____ 166

Uporaba lastne knjižnice ____ 172

Abstraktni razredi in abstraktne metode ____ 179

Abstraktni razredi in virtualne metode ____ 187

Večokenske aplikacije ____ 188

Izdelava novega obrazca ____ 189

Odpiranje nemodalnih obrazcev ____ 189

Odpiranje pogovornih oken - modalnih obrazcev ____ 190

Vključevanje že obstoječih obrazcev v projekt ____ 192

Odstranjevanje obrazca iz projekta ____ 192

Sporočilno okno AboutBox ____ 192

Dostop do polj, metod in gradnikov drugega obrazca ____ 194

Garbage Collector ____ 202

Uporaba Garbage Collectorja in upravljanje s pomnilnikom ____ 202

Destruktorji in Garbage Collector ____ 202

Kako deluje Garbage Collector ____ 203

Upravljanje s pomnilnikom ____ 203

Using stavek ____ 204

MDI – Multi Document Interface (vmesnik z več dokumenti) ____ *215*

Tiskalnik ____ *227*

Tiskanje enostranskega besedila ____ 229

Tiskanje besedila, ki obsega poljubno število strani ____ 232

Tiskanje slik ____ 234

Povzetek ____ *243*

Seznam krvodajalcev ____ *243*

DRŽAVE ____ 253

Podatkovna skladišča (baze) in upravljanje s podatki ____ *253*

Izdelava baze podatkov ____ 253

Izdelava novega projekta, ki dela nad obstoječo bazo podatkov – uporaba čarovnika ____ 259

Gradnik BindingNavigator in gradnik TableAdapterManager ____ 261

Programska uporaba ADO.NET (brez čarovnika) ____ 268

V brazcu FZnamka moramo zapisati le odzivno metodo gumba *Shrani*. Gumbu *Prekliči* smo le nastavili lastnost *DialogResult* na *false*.. ____ 284

Povezava med dvema tabelama: prikaz le določenih podatkov iz tabele ____ 285

Transakcije ____ 292

Povzetek ____ *295*

Države ____ *297*

PRIPRAVA NAMESTITVENEGA PROGRAMA, TESTIRANJE ____ 311

XCopy ____ *311*

ClickOnce ____ *312*

Setup (namestitveni) program ____ *316*

DOKUMENTIRANJE ____ **317**

Dokumentiranje razredov ____ *317*

LITERATURA IN VIRI ____ **321**

KAZALO SLIK:

Slika 1: Okno z rešitvijo in projekti v njej. 15

Slika 2: Ustvarjanje novega projekta. 17

Slika 3: Vnos imena projekta in mape, kjer bo le-ta shranjen. 17

Slika 4: Razvojno okolje. 19

Slika 5: Preklapljanje med oblikovnim pogledom (*Design*) in pogledom s kodo obrazca. 19

Slika 6: Prvi okenski program. 21

Slika 7: Urejanje lastnosti gradnikov. 22

Slika 8: Seznam vseh gradnikov na obrazcu. 22

Slika 9: V urejevalniškem oknu imamo lahko hkrati odprta oba pogleda na obrazec: pogled *Code View* in pogled *Design View*. 23

Slika 10: Zvezdica na vrhu okna pomeni, da zadnje stanje projekta še ni shranjeno. 23

Slika 11: Gradniki, ki jih potrebujemo na obrazcu *Kitajski horoskop*. 25

Slika 12: Pozdravno sporočilo projekta *Kitajski horoskop*. 26

Slej ko prej bomo želeli napisati projekt, v katerega se bo moral uporabnik preko nekega obrazca najprej prijavit (glej Slika 13). 28

Slika 14: V Slikarju pripravimo ozadje obrazca. 28

Slika 15: Gradniki na prijavnem obrazcu. 29

Slika 16: Sporočilno okno z nastavitvami na prijavnem obrazcu. 31

Slika 17: Gradniki na anketnem obrazcu. 32

Slika 18: Anketni obrazec z vnesenimi podatki. 36

Slika 19: Sprememba imena odzivnega dogodka. 42

Slika 20: Lastnosti drugega parametra odzivnih metod. 43

Slika 21: Gradniki in njihova imena na obrazcu *FPreverjanje*. 44

Slika 22: Če se v delujočem programu z miško postavimo na gumb *Info*, se nam prikaže oblaček z ustrezno informacijo! 48

Slika 23: Parameter *sender* in operator *as*. 49

Slika 24: Parameter *sender* in operator *is*. 49

Slika 25: Hitrostno klikanje. 51

Slika 26: Kontrola uporabnikovih vnosov. 54

Slika 27: Če so vnosi napačni (oz. vnosa ni), se ob gradniku pojavi ikona z opozorilom. 56

Slika 28: Osnovna uporaba sporočilnega okna *MessageBox*. 57

Slika 29: Sporočilno okno z dvema parametroma tipa string. 57

Slika 30: Uporaba parametra *MessageBoxButtons* v sporočilnem oknu....59

Slika 31: Sporočilno okno z ikono. 60

Slika 32: Sporočilno okno s tremi gumbi, ikono in izbranim aktivnim gumbom. 61

Slika 33: Sporočilno okno z desno poravnavo teksta. 62

Slika 34: Prikaz gumba *Help* (Pomoč). 63

Slika 35: Prikaz gumba in datoteke s pomočjo. 64

Slika 36: Ustvarjanje glavnega menija – na obrazec postavimo gradnik *MenuStrip*. 65

Slika 37: Glavni meni s standardnimi opcijami. 66

Slika 38: Ročno oblikovanje menija s pomočjo urejevalnika menija. 66

Slika 39: Pri ustvarjanju menija lahko izbiramo med tremi možnostmi. 67

Slika 40: *Pop Up* meni za oblikovanje menija. 67

Slika 41: Ustvarjanje lebdečega menija. 69

Slika 42: Oblikovanje orodjarne. 69

Slika 43: Gradnik *StatusStrip* s tremi predalčki. 70

Slika 44: Pogovorno okno *ColorDialog*. 72

Slika 45: Ustvarjanje lastne barve palete v pogovornem oknu *ColorDialog*. 72

Slika 46: Gradniki na obrazcu za prikaz pogovornega okna *FolderBrowserDialog*. 74

Slika 47: *FolderBrowserDialog* z nekaterimi programskimi prednastavitvami. 76

Slika 48: Pogovorno okno *FontDialog* za izbiro pisave. 77

Slika 49: Pogovorno okno *OpenFileDialog*....79

Slika 50: Obrazec za predvajanje glasbe in videa. 81

Slika 51: Gradniki *Visual C#* urejevalnika....84

Slika 52: Prikaz začetnega obrazca projekta *DataGridViewDemo*. 89

Slika 53: Prikaz obrazca v primeru ažuriranja podatkov. 90

Slika 54: Ustvarjanje imen stolpcev gradnika *DataGridView*. 91

Slika 55: Nastavitev v oknu *DataGridView Tasks*. 92

Slika 56: Gradniki projekta *DataGridViewDemo*. 93

Slika 57: Prikaz vsebine tipizirane zbirke v gradniku *DataGridView*....100

Slika 58: Sporočilna okna s prikazom različnih kombinacij pritiska tipke Q. 102

Slika 59: Gradniki na obrazcu *FAlkohol*. 104

Slika 60: Obvestilo o napačnem vnosu! Slika 61: Rezultat pravilnega vnosa. 108

Slika 62: Ustvarjanje novega zavihka gradnika TabControl. 110

Slika 63: Objektii na prvem zavihku gradnika *TabControl*. 111

Slika 64: Objekti na drugem zavihku gradnika *TabControl*. 113

Slika 65: Objekta na tretjem zavihku gradnika *TabControl*....114

Slika 66: Sestavni deli labirinta. 116

Slika 67: Gradniki na obrazcu *FSpomin*. 119

Slika 68: Izdelava stolpcev v gradniku *DataGridView* in nastavitev lastnosti. 121

Slika 69: Končni izgled programa *Spomin*!....125

Slika 70: Gradniki na obrazcu za prikaz štoparice. 126

Slika 71: Prikaz delovanja štoparice. 126

Slika 72: Gradniki na obrazcu f*UgibajBesedo* 129

Slika 73: Ugibanje besede....134

Slika 74: Pomoč pri dinamičnem ustvarjanju okenskega dogodka. 137

Slika 75: Razvojno okolje nam ponudi možnost zapisa ogrodja odzivnega dogodka. 137

Slika 76: Dinamično ustvarjeni okenski gradniki na obrazcu. 138

Slika 77: Program *Tri v vrsto* med igro. 141

Slika 78: Gradniki obrazca *Tri v vrsto*. 142

Slika 79: Začetna velikost obrazca *Tri v vrsto*....142

Slika 80: Predstavitev dveh objektov razreda *Tocka*. 151

Slika 81: Predstavitev dveh objektov razreda *Krog*. 152

Slika 82: Predstavitev dveh objektov razreda *Valj*. 153

Slika 83: Gradniki na obrazcu Videoteka. 157

Slika 84: Gradniki bazičnega obrazca *FObrazec*. 162

Slika 85: Nov gradnik: utripajoča oznaka....169

Slika 86: Sporočilno okno, ki se pokaže, če skušamo pognati dll. 172

Slika 87: Nova gradnika *NumberBox* in *Gumb* v oknu *Toolbox*. 173

Slika 88: Kalkulator....174

Slika 89: Obrazec za izračun obresti. 180

Slika 90: Miselna igra trije sodi! 183

Slika 91: Spročilno okno *AboutBox*. 193

Slika 92: Koda obrazca *AboutBox*....193

Slika 93: Okno za nastavljanje lastnosti projekta. 194

Slika 94: Glavni obrazec kviza. 196

Slika 95: Obrazec *FKviz*. 197

Slika 96: Glavni obrazec projekta *Evidenca članov športnega društva*....205

Slika 97: Članski obrazec *FClanskiObrazec*....207

Slika 98: Obrazec za pregled in urejanje članov športnega društva. 207

Slika 99: *MDI* obrazec in otroška okna. 216

Slika 100: Starševski obrazec - *MDI Parent*......216

Slika 101: Otroški obrazec - *MDI Child*. 217

Slika 102: Modalno okno s pomočjo!......218

Slika 103: Glavni obrazec *MDI* aplikacije. 221

Slika 104: Lebdeči meni na otroškem obrazcu! 221

Slika 105: Otroški obrazec - brskalnik. 222

Slika 106: Dialog za izbiro tiskalnika in dialog za nastavitev strani izpisa. 228

Slika 107: Obrazec za tiskanje pisma......229

Slika 108: Predogled tiskanja......232

Slika 109: Obrazec za tiskanje večstranskega besedila. 232

Slika 110: Seznam gradnikov projektu *Menjalnica*. 235

Slika 111: Projekt *Menjalnica* v uporabi......242

Slika 112: Projekt *Menjalnica*: primer izpisa na tiskalniku......242

Slika 113: Glavni obrazec projekta *Seznam krvodajalcev*......245

Slika 114: Obrazec za Vnos/Ažuriranje podatkov krvodajalca. 246

Slika 115: Primer izpisa seznama krvodajalcev. 252

Slika 116: Izdelava nove baze podatkov......254

Slika 117: Okno *Solution Explorer* in v njem objekt tipa *DataSet* z imenom *PisateljiDataSet*......255

Slika 118: Okno *DataBase Explorer*......255

Slika 119: Ustvarjanje polj v tabeli *Pisatelji*. 255

Slika 120: Okno *Column Properties*......256

Slika 121: Vnos podatkov v tabelo *Pisatelji*. 256

Slika 122: Tabela *Pisatelji* v gradniku *DataSet*. 257

Slika 123: Povezava gradnika *DataGridView* s podatkovnim izvorom – tabelo v bazi podatkov. 257

Slika 124: V oknu *Data Source Configuration Wizard* izberemo tabele, ki jih bomo potrebovali. 259

Slika 125: *Dataset* z vključenimi tabelami iz baze *NabavaSQL*. 260

Slika 126: Okno *Add Connection*. 260
Slika 127: Okno *Data Sources*. 261
Slika 128: Gradnika *BindingNavigator* in *DataGridView*. 262
Slika 129: Gradnik *BindingNavigator*. 262
Slika 130: *BindingNavigator* z vsemi standardnimi urejevalniškimi gumbi. 262
Slika 131: Dodajanje novih gumbov v *BindingNavigator*....263
Slika 132: Gradnik *TableAdapterManager*. 264
Slika 133: Povezovanje stolpca *Obdobje* s podatkovnim izvorom, to je s tabelo *Obdobje*. 266
Slika 134: Okni *DataBase Explorer* in *Data Sources*, ter izbira vrste prikaza podatkov. 267
Slika 135: Prikaz vsebine tabele iz baze podatkov v podrobnem načinu (*Details*). 267
Slika 136: Glavni obrazec projekta *RabljenaVozila* 272
Slika 137: Obrazec za vnos novega avtomobila in hkrati za ažuriranje podatkov o avtomobilu. 275
Slika 138: Obrazec za prikaz tabele Znamke. 281
Slika 139: Obrazec za vnos oz. ažuriranje znamke vozila. 284
Slika 140: Povezava gradnika *ComboBox* s tabelo *Dobavitelji* in *DataGridView* s tabelo *Artikli*. 285
Slika 141: Obrazec za izdelavo poizvedb iz baze *JadralciSQL*....288
Slika 142: *Dataset* z dvema tabelama. 292
Slika 143: Obrazec za prikaz transakcij. 293
Slika 144: Testni podatki v tabelah *Kontinenti* in *Drzave* baze *DrzaveSQL*. 297
Slika 145: Glavni obrazec projekta *DržaveNaSvetu*. 298
Slika 146: Podobrazec za dodajanje nove države in za ažuririranje podatkov o izbrani državi. 299
Slika 147: Obrazec za prikaz, urejanje, brisanje in dodajanje kontinentov. 306
Slika 148: Obrazec dodajanje/ažuriranje kontinenta. 309
Slika 149: Ustvarjanje namestitvenih datotek na *ftp* strežniku. 313
Slika 150: Okno v katerem določimo, kako bo uporabnik namestil svojo aplikacijo. 313
Slika 151: Okno v katerem določimo, ali bo za namestitev potrebna prijava ali ne! 314

Slika 152: Namestitveno okno naše aplikacije. 314

Slika 153: Varnostno opozorilo pred namestitvijo! 315

Slika 154: Projekt *XMLDokumentacija*. 318

Slika 155: Sistem *IntelliSense* poleg imena metode prikaže tudi okno z našo dokumentacijo. 319

Slika 156: Izpis *XML* dokumentacije za parameter metode. 320

Slika 157: Izpis dokumentacije o našem razredu. 320

KAZALO TABEL

Tabela 1: Seznam bližnjic v razvojnem okolju. 21

Tabela 2: Najpomembnejše metode obrazca. 40

Tabela 3: Najpomembnejši dogodki obrazca. 41

Tabela 4: Metode razreda *object*. 43

Tabela 5: Lastnosti gradnikov na obrazcu z imenom *FPreverjanje*. 45

Tabela 6: Gradniki in njihove lastnosti. 50

Tabela 7: Gradniki na obrazcu *FValidacija*. 54

Tabela 8: Vrednosti naštevnega tipa *MessageBoxButtons.* 58

Tabela 9: Vrednosti tipa *DialogResult*. 58

Tabela 10: Vrednosti naštevnega tipa *MessageBoxIcon*. 60

Tabela 11: Tabela možnih nastavitev v oknu *MessageBox* 61

Tabela 12: Najpomembnejše lastnosti pogovornega okna *ColorDialog*. 71

Tabela 13: Glavne lastnosti pogovornega okna *FolderBrowserDialog*. 73

Tabela 14: Najpomembnejše lastnosti pogovornega okna *FontDialog*. 77

Tabela 15: Lastnosti pogovornega okna *OpenFileDialog*. 78

Tabela 16: Tabela lastnosti pogovornega okna *SaveFileDialog*. 80

Tabela 17: Lastnosti stolpcev gradnika *DataGridView*. 92

Tabela 18: Dogodki tipkovnice. 101

Tabela 19: Lastnosti gradnikov obrazca *FAlkohol*. 105

Tabela 20: Dogodki miške. ____ 110

Tabela 21: Lastnosti gradnikov obrazca *FSpomin*. ____ 120

Tabela 22: Lastnosti gradnikov na obrazcu f*UgibajBesedo*. ____ 130

Tabela 23: Lastnosti gradnikov bazičnega obrazca. ____ 164

Tabela 24: Najpomembnejše lastnosti in metode razreda *PrintPageEventArgs*. ____ 228

Tabela 25: Tabela najpogostejših elementov XML dokumentacije. ____ 317

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.

TRI V VRSTO

Igra ***Tri v vrsto*** je igra za dva igralca, ki polagata ploščice dveh barv na mrežo 3 x 3 polj (ali pa rišeta križce in krožce na karo papirju). Zmaga tisti, ki mu prvemu uspe postaviti v vrsto ali diagonalo tri svoje ploščice.

Radi bi napisali program, ki bo igralcema omogočal igranje te enostavne igrice. V ta namen bomo šli preko vrste gradnikov. Slednje bomo tudi ilustrirali z različnimi zgledi. Spoznali bomo osnove razvojnega okolja, pojasnili kako gradimo aplikacijo s pomočjo obrazca in grafičnih gradnikov, ki jih polagamo na obrazce, ter kako pišemo odzivne dogodke. Naučili se bomo uporabljati enega od najkompleksnejših gradnikov *DataGridView* in dinamično ustvarjati okenske gradnike in njihove dogodke. Obenem bomo ponovili, kako pišemo lastne metode in kakšen je pomen varovalnih blokov.

MICROSOFT Visual C# 2010 - OSNOVNI POJMI: Rešitve (Solutions) in projekti (projects)

Microsoft Visual Studio je razvojno okolje (IDE – *Integrated Development Enviromnment*), ki omogoča razvoj konzolnih, namiznih in spletnih aplikacij na platformah Microsoft Windows, *Windows Mobile*, *Windows CE*, *.NET Framework*, *.NET Compact Framework* in *Microsoft Silverlight.* V okviru razvojnega okolja lahko uporabljamo različne programske jezike: Visual C#, Visual C++ , J# ali pa Visual Basic.

Glede na potrebe programerjev Microsoft ponuja kar nekaj verzij razvojnega okolja: *Professional* (namenjen posameznikom za osnovne razvojne naloge), *Premium* (namenjen posameznikom in projektnim skupinam za razvoj aplikacij) in *Ultimate* (obširna množica orodij ki pomagajo razvijalcem in skupinam izdelati visoko kvalitetne projekte od oblikovanja do instalacije). Poleg tega Microsoft ponuja še kar nekaj drugih sorodnih orodij za spodbujanje kreativnosti in izdelavo zahtevnih in sodobnih rešitev.

Najzanimivejši, ker so dostopni brezplačno, pa so za široke množice prav gotovo produkti *Visual Studio Express*. Odločimo se lahko za različice, ki podpirajo katerega koli od prej naštetih jezikov. V tej literaturi bomo uporabljali najnovejšo različico *Visual C# 2010 Express* (kratica ***VCE***).

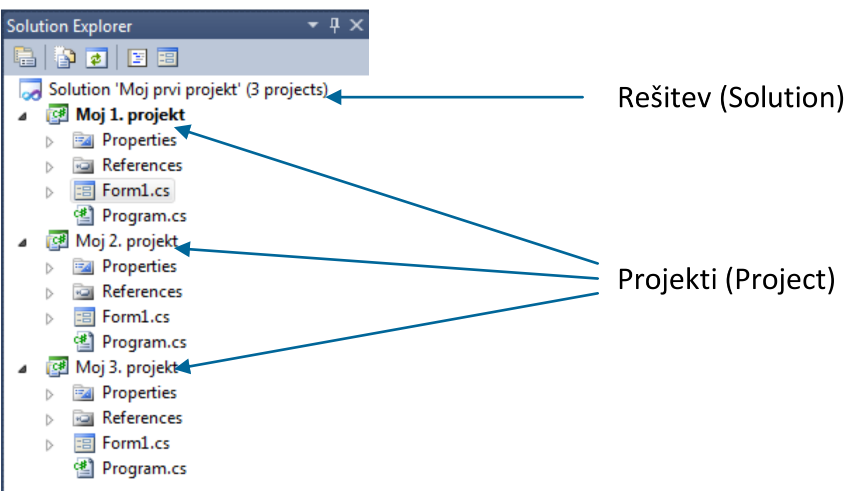
Največja organizacijska enota v okolju *MS Visual Studio* in tako tudi v *Visual C# 2010 Express* je **rešitev** - **Solution**. Posamezna rešitev obsega *celotno* delo, ki se tiče nekega problema. Vsebuje

enega ali več projektov (**Project)**, a običajno je projekt en sam. Projekt vsebuje izvorne datoteke in še vse ostale datoteke (npr. slike) , ki jih potrebuje prevajalnik za izdelavo izvršilne datoteke (.exe) oz. knjižnjice (.dll). Rešitev, ki vsebuje več projektov, je primerna za velike projekte, ki jih razvija več razvijalcev hkrati. Le-tem je tako omogočeno neodvisno delo na projektih, ki skupaj tvorijo ustrezno rešitev.

Projekt vsebuje izvorne datoteke aplikacije, ki jih prevajalnik (*compiler*) prevede v objektne datoteke vrste *.obj*, povezovalnik (*linker*) pa le-te poveže v *izvedbeno* datoteko aplikacije vrste *.exe* ali *dinamično* knjižnico vrste *.dll*. Oba postopka skupaj se imenujeta gradnja projekta (*building*).

Vsaka izvedba neke aplikacije zahteva ločen projekt. Tudi če želimo npr. za isto aplikacijo zgraditi izvedbeno datoteko vrste *.exe* in izvedbeno datoteko vrste dinamične knjižnice *.dll*, moramo uporabiti za vsako od njiju ločen projekt.

Okno *Solution Explorer* prikazuje rešitev, projekte in seznam njihovih datotek:



Slika 1: Okno z rešitvijo in projekti v njej.

Zapomnimo si:

- Posamezen odprt primerek *Visual C# 2010 Express* v operacijskem sistemu Windows obsega le eno rešitev.
- Rešitev obsega enega ali več projektov ter morebitne odvisnosti med njimi.
- Posamezen projekt lahko pripada eni ali pa več rešitvam.
- Posamezni projekti v neki rešitvi so lahko tudi v različnih programskih jezikih (takih, ki jih podpira razvojno orodje).

Razvojno orodje običajno ustvari (razen če spremenimo nastavitve) za vsak projekt ločeno mapo na disku. V tej mapi sta tudi podmapi *"obj"* ter *"bin"***,** v katere *VCE* shranjuje datoteke, ki sestavljajo projekt.

VCE uporablja dva tipa datotek (*.sln* in ***.suo***) za shranjevanje nastavitev, ki ustrezajo neki rešitvi. Ti dve datoteki, ki ju s skupnim imenom imenujemo *rešitev - solution files*, izdela *Solution Explorer*. Opremi ju z vsemi informacijami in podatki, ki so potrebni za prikaz grafičnega vmesnika in upravljanje z datotekami našega projekta. Razvijalcu projekta se tako ob odpiranju razvojnega okolja ni potrebno vsakič ukvarjati z okoljskimi nastavitvami.

Rešitev (*Solution*) je torej sestavljena iz dveh datotek:

- *.sln (Visual Studio Solution)*, ki vsebuje popoln opis o vsebini rešitve. Organizira projekt in njegove komponente v skupno rešitev in zagotovi ustrezno okolje s potrebnimi referencami in njihovim položajem na mediju - disku; datoteko lahko uporablja več razvijalcev aplikacije.
- ***.****suo (Solution User Options)*, ki vsebuje podatke o pozicijah oken. Datoteka je specifična za posameznega razvijalca, hrani pa vse potrebne nastavitve, zato da jih pri ponovnem odpiranju projekta ni potrebno nastavljati ponovno.

Datoteka *.sln* je nepogrešljiva, datoteka *.suo* pa je pogrešljiva: rešitev lahko zgradimo tudi brez nje.

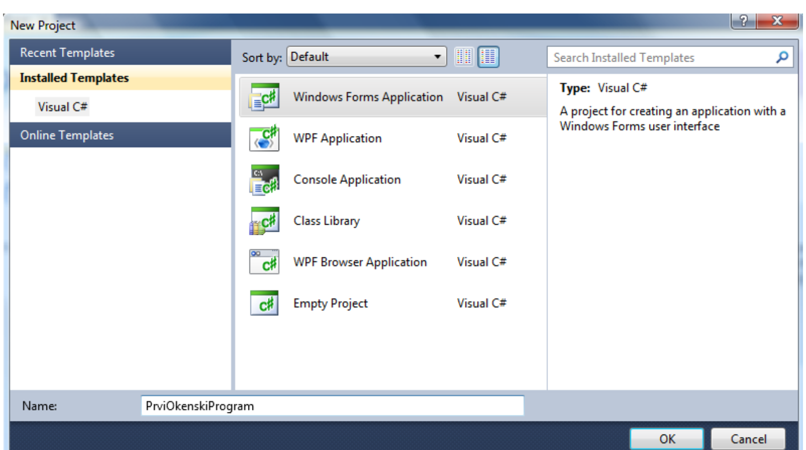
Okenske aplikacije (Windows Forms) v Visual C# 2010 Express Edition

Razvojno orodje **Visual C#** vsebuje tudi vsa potrebna orodja za razvoj okenskih aplikacij. S pomočjo **Visual Designer**-ja lahko ustvarimo uporabniški vmesnik, ki temelji na t.i. obrazcih (*Forms*). Enostavni projekti lahko vsebujejo en sam obrazec, kompleksnejši pa celo množico različnih obrazcev, ki jih uporabnik po potrebi odpira in zapira. Pri gradnji aplikacije na obrazce postavljamo gradnike, **Visual C#** pa sam generira programsko kodo, ki ustreza uporabniškemu vmesniku, ki ga oblikujemo. Največja razlika med gradnjo konzolnih in vizuelnih aplikacij pa je v tem, da smo pri konzolnih aplikacijah v glavnem delali z eno samo datoteko, v katero smo pisali kodo, pri vizualnih aplikacijah pa je datotek več, vsaka od njih pa ima točno določen pomen. Pri gradnji projektov se bomo opirali na predznanje, ki smo ga pridobili v preteklosti: zanke, tabele, metode, strukture, razredi in objekti, datoteke in še kaj. Vse o teh podatkovnih tipih in številne vaje lahko najdete npr. v datotekah

- http://uranic.tsckr.si/C%23/C%23.pdf,
- http://uranic.tsckr.si/C%23/Prakticno%20programiranje.pdf,
- http://uranic.tsckr.si/VISUAL%20C%23/VISUAL%20C%23.pdf,
- http://uranic.tsckr.si/C%23/.

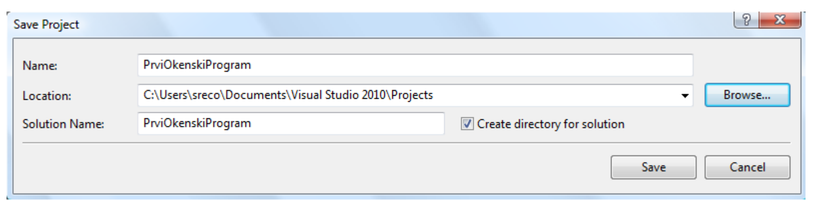
Ustvarjanje osnovnih okenskih aplikacij v okolju Visual C#

Za izdelavo prve okenske aplikacije odpremo meni *File → New Project*. Odpre se novo okno **New Project** v katerem izberemo opcijo **Windows Forms Application**. Na dnu okna, v polje *Name*, napišimo še ime naše prve okenske aplikacije, npr. *PrviOkenskiProgram* in kliknemo gumb *OK*. Na ekranu se prikaže celotno razvojno okolje za gradnjo novega projekta. Še preden naredimo karkoli, novo začeti projekt takoj shranimo. Izberemo *File→Save All* (ali pa kliknemo ikono *Save All* v orodjarni).



Slika 2: Ustvarjanje novega projekta.

Odpre se pogovorno okno *Save Project* za shranjevanje projekta. Ime projekta (*Name*) je že zapisano, izbrati ali potrditi moramo le še lokacijo (*Location*), kjer bo naš projekt shranjen. Privzeta nastavitev ja kar mapa Project znotraj mape *Visual Studio 2010*, seveda pa se lahko s klikom na gumb *Browse* prestavimo v drugo mapo ali pa kjerkoli ustvarimo novo mapo, v kateri bo naš projekt.



Slika 3: Vnos imena projekta in mape, kjer bo le-ta shranjen.

Čeprav smo v prejšnjem odstavku zapisali, da v polje *Name* vnesemo ime naše prve okenske aplikacije, to ne drži povsem. Dejansko gre za ime imenskega prostora (*namespace*) znotraj katerega bomo ustvarili nov projekt. Dejansko ime projekta, ki ga delamo znotraj določenega imenskega prostora, bomo vnesli šele ob kliku na *SaveAll* v razvojnem okolju. In kakšen je pomen pojma imenski prostor: zaenkrat bo dovolj da vemo, da kadar različni razvijalci delajo na istem projektu, uporabljajo vsak svoj imenski prostor za razvoj. Znotraj le-tega, jim potem ni potrebno skrbeti za probleme, ki lahko nastanejo zaradi podvajanja imen (npr. imen razredov). Mi bomo v naših projektih običajno imenski prostor poimenovali enako kot projekt.

Razvojno orodje nam ponuja tudi možnost, da je ime rešitve (*Solution Name*) drugačno od imena programa, a v naših projektih imena rešitve zaenkrat ne bomo spreminjali. Kljukica v polju *Create Directory for solution* omogoča, da bo v mapi, ki smo jo izbrali ali ustvarili za naš projekt (*Location*), ustvarjena nova mapa z imenom našega projekta, v tej mapi pa bodo vse datoteke, ki sestavljajo naš projekt.

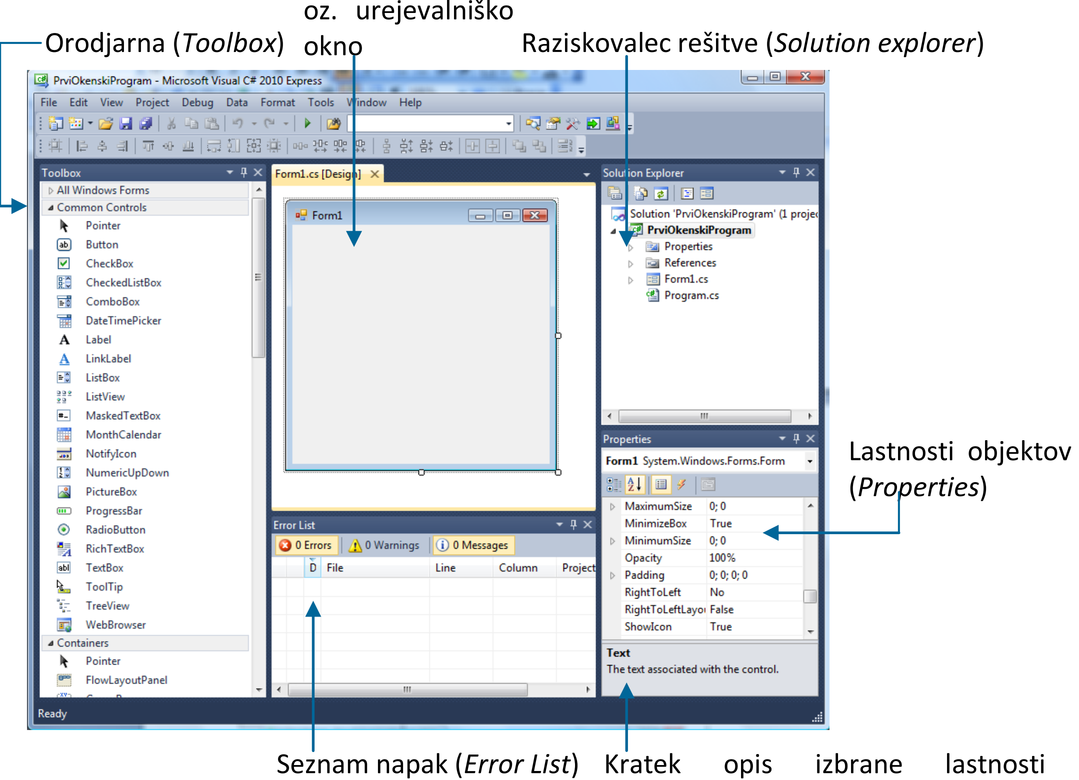
Projekt dokončno shranimo s klikom na gumb *Save*. Pri vsakem prevajanju se bo novo stanje zapisalo v izbrano mapo. Obenem se bo vselej ustvaril tudi izvršilni program (datoteke s končnico *.exe* - *executable file*), ki nastane kot končni rezultat našega projekta. Ta datoteka se nahaja v mapi *imeProjekta**bin**Debug* znotraj mape, v kateri je naš projekt. Če želimo našo že izdelano aplikacijo prenesti na drug računalnik, je potrebno le skopirati izvršilno datoteko in le-to prenesti na drug računalnik (na katerem pa mora biti nameščen *Microsoftov Framework* – to je programska oprema, ki jo namestimo na računalnik na katerem teče operacijski sistem *Windows*, vsebuje pa številne knjižnjice z rešitvami za uporabnike in razvijalce aplikacij).

Posvetimo se sedaj najprej opisu razvojnega okolja. Po vnosu imena projekta in shranjevanju imamo na vrhu razvojnega okolja t.i. *glavno okno*, ki vsebuje vrstico z meniji, pod njo pa orodjarno s hitrimi gumbi za realizacijo posebno pomembnih opravil. Pod glavnim oknom je odprt nov prazen obrazec (*Windows Form*) v oblikovalskem pogledu (*Design View*), okoli njega pa nekaj odprtih oken, ki nam pomagajo pri gradnji projekta.

Število oken, ki so prikazana v razvojnem okolju je odvisno od nastavitev. Zgornja slika prikazuje najpogosteje odprta okna, pri čemer pa lahko katerokoli od oken tudi minimiziramo oz. zopet prikažemo. Če je npr okno *Toolbox* skrito, kliknemo na *Toolbox* na levem robu ekrana in ko se prikaže, ga lahko s klikom na priponko fiksiramo na tem delu ekrana. Če pa je npr. okno *Toolbox* povsem zaprto, ga ponovno odpremo z izbiro *View* → *Other Windows*→*Toolbox*. Enako pravilo velja tudi za okni *Solution Explorer* in *Properties,* pa seveda za vsa ostala okna (razen glavnega, ki je odprto ves čas).

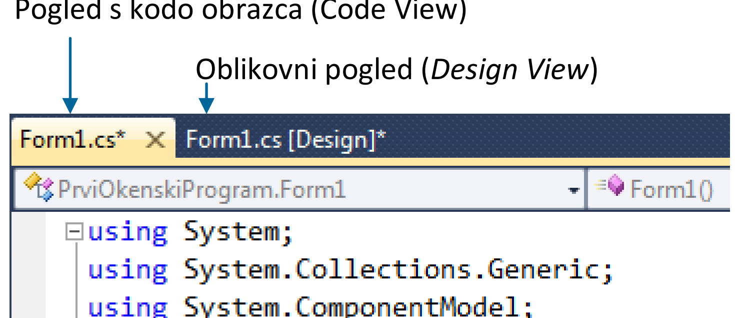
Okna, ki sestavljajo razvojno okolje imajo naslednji pomen:

- *Toolbox*: sestavljena je iz več palet z gradniki. Projekt v Visual C# namreč gradimo tako, da na obrazce postavljamo gradnike. Na paleti *All Windows Forms* so zbrani prav vsi gradniki po abecednem redu, na ostalih paletah pa le gradniki, ki po neki logiki spadajo skupaj: na paleti *Common Controls* so tako zbrani najpogosteje uporabljeni gradniki, na paleti *Menus & Toolbars* gradniki za delo z meniji, orodjarno, ipd. Posamezno skupino gradnikov odpremo s klikom miške na znak **+** pred imenom skupine, zapremo pa s klikom na znak **–** pred imenom skupine.



Slika 4: Razvojno okolje.

- Obrazec oz. urejevalniško okno: privzeto je v tem oknu slika obrazca, na katerega bomo polagali gradnike. S tipko *F7* (ali pa klikom na *View→Code*) lahko prikažemo datoteko, v katero bomo pisali kodo (spremenljivke, objekte, metode, odzivne dogodke), ki pripadajo temu obrazcu. V oknu *Code/Design View* se pokaže nov zavihek. Če sedaj kliknemo zavihek (*Tab*), ki ima dodano besedico *[design]* se vrnemo nazaj na oblikovni pogled obrazca.



Slika 5: Preklapljanje med oblikovnim pogledom (*Design*) in pogledom s kodo obrazca.

S tipkama *Shift+F7* (oziroma *View→Designer*) preklopimo nazaj na vizuelni del obrazca.

- Solution Explorer: raziskovalec rešitve je okno, v katerem so prikazane vse mape in datoteke znotraj našega projekta. Pomen posameznih datotek in map bomo spoznavali kasneje, ko bomo ustvarjali projekte.

- Properties: okenske (vizuelne) aplikacije gradimo tako, da na obrazec postavljamo gradnike, ki jih jemljemo iz okna *Toolbox*. V oknu *Properties* so prikazane lastnosti in dogodki tistega gradnika na obrazcu, ki je trenutno izbran (tudi obrazec je gradnik). Med njimi je tudi ime *(name)*, ki določa ime posameznega gradnika. Z () je označeno, da ne gre za lastnost z imenom name. S pomočjo tega imena se v projektu na posamezne gradnike tudi sklicujemo. Ime mora biti enolično na nivoju celotnega projekta. Pri tem nam je razvojno okolje v veliko pomoč, saj nam za vsak nov gradnik predlaga privzeto ime. *Visual C#* poimenuje imena gradnikov tako, da jim zaporedoma dodeli imena z dodano številko na koncu (*label1*, *label2*, *label3*,… ali pa *button1*, *button2*, …). Imena gradnikov lahko poljubno spremenimo (to je pri obsežnejših projektih celo priporočljivo oz. skoraj nujno, saj sicer izgubimo pregled nad gradniki), a pri tem je potrebno paziti, da se imena ne podvajajo, sicer bomo dobili obvestilo o napaki. Kadar poimenujemo gradnike s svojim imenom, je priporočljivo, da uporabljamo zveneča imena: npr *gumbVnos*, *gumbZapri*, *gumbAzuriraj* … Na začetku v naših projektih privzetih imen gradnikov ne bomo spreminjali, a le v primerih, ko gradnikov istega tipa ne bo veliko.

- *Error List:* okno s seznamom napak. Ko namreč nek projekt prevedemo, so v njem lahko napake, ki so prikazane v tem oknu. Klik na posamezno vrstico v tem oknu nas postavi v urejevalnik kode na mesto, kjer se ta napaka nahaja.

Razvojno okolje nam ponuja tudi druga okna, ki nam pomagajo pri gradnji projekta, ki pa jih zaenkrat ne bomo potrebovali.

Če se v oknu *Toolbox* z miško zapeljemo čez poljuben gradnik in za trenutek počakamo, se v okvirčku izpiše osnovni namen tega gradnika.

Več o posameznih oknih bomo spoznali v nadaljevanju, ob ustvarjanju projektov.

V naslednji tabeli so zbrane osnovne bližnjice za določena opravila v razvojnem okolju, ki jih lahko naredimo s pomočjo tipkovnice.

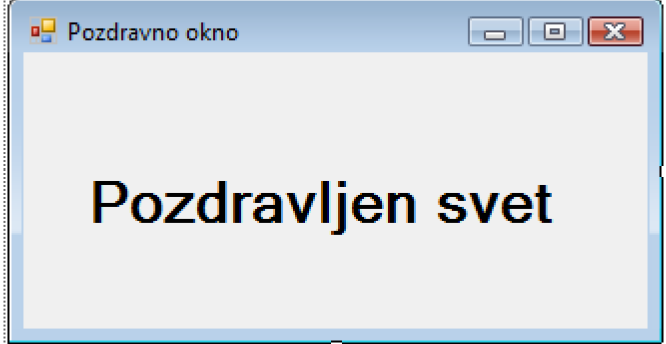
Bližnjica	Razlaga
F7	Preklop med pogledom *Design View* in *Code View*.
Shift+F7	Preklop med pogledom *Code View* in *Design View.*
F9	Nastavitev prekinitvene točke.
F12	Skok na definicijo spremenljivke, objekta ali metode.
Shift+F12	Iskanje vseh referenc metode ali spremenljivke.

Ctrl+M, Ctrl+M	Razširi / Skrij del kode (npr. vsebino metode) v urejevalniku.
Ctrl+K, Ctrl+C	Zakomentiraj označeni del kode v urejevalniku.
Ctrl+K, Ctrl+U	Odkomentiraj označeni del kode v urejevalniku.
Shift+Alt+Enter	Preklop med celostranskim pogledom na urejevalniško okno in normalnim pogledom.
Ctrl+I	Zaporedno iskanje določene besede (spremenljivke, metode, ..); vse enake besede v oknu se označijo.

Tabela 1: Seznam bližnjic v razvojnem okolju.

Nadaljujmo sedaj z gradnjo našega prvega okenskega programa, poimenovanega *PrviOkenskiProgram*. Najprej kliknimo na obrazec, se postavimo v okno *Propeties*, izberemo lastnost *Text* in vpišemo besedilo "Pozdravno okno". Vnos potrdimo z <*Enter*> oz. ponovnim klikom na obrazec. Privzeto ime obrazca (lastnost *Name*) je *Form1* in ga ne bomo spreminjali. Na obrazec sedaj postavimo naš prvi gradnik. V oknu *Toolbox* izberimo (kliknimo) gradnik *Label* – oznaka (nahaja se na paleti *Common Controls*), nato pa kliknimo na površino obrazca na mesto, kamor želimo postaviti ta gradnik. Dobil je privzeto ime *label1*. Vsak gradnik, ki ga postavimo na obrazec ima na začetku privzeto ime in lastnosti, ki pa jih lahko spremenimo v oknu *Properties*. Spomnimo se, da ga najdemo v pregledovalniku lastnosti pod (name). Gradnik je namenjen prikazu poljubne informacije na obrazcu. Pogosto ga tudi postavimo ob kak drug gradnik in vanj zapišemo besedilo, s katerim opišemo namen tega gradnika.

Prepričajmo se, da je gradnik *label1* na obrazcu izbran in se preselimo v okno *Properties* (če le-to ni odprto, ga odpremo z *View→Other Windows→Properties Window*). Med lastnostmi najprej poiščimo lastnost *Text* in jo spremenimo v "*Pozdravljen svet*". Poiščimo še lastnost *Font*, kliknimo na tripičje in v pogovornem oknu izberemo določeno vrsto pisave, stil in velikost. V urejevalniškem oknu nato izberemo obrazec (na njegovih robovih se prikažejo kvadratki) in ga ustrezno pomanjšamo (razširimo), da dobimo približno takole sliko.



Slika 6: Prvi okenski program

Gradnikom na obrazcu lahko poljubno spreminjamo velikost. Izberemo gradnik, ki ga želimo povečati/pomanjšati (tako da nanj kliknemo…) , nato pa ga z miško raztegnemo/skrčimo na želeno velikost. Če je na obrazcu več gradnikov, je možna njihova poravnava tako, da nek gradnik približamo drugemu in prikažeta se vodoravna in navpična črta, ki omogočata natančno poravnavo.

Ker smo v oknu *Design View* nazadnje kliknili na obrazec (tudi obrazec je gradnik), so v oknu *Properties* sedaj prikazane lastnosti obrazca. Vsak od gradnikov ima cel kup lastnosti, posamezne izmed njih pa bomo spoznavali v nadaljevanju.

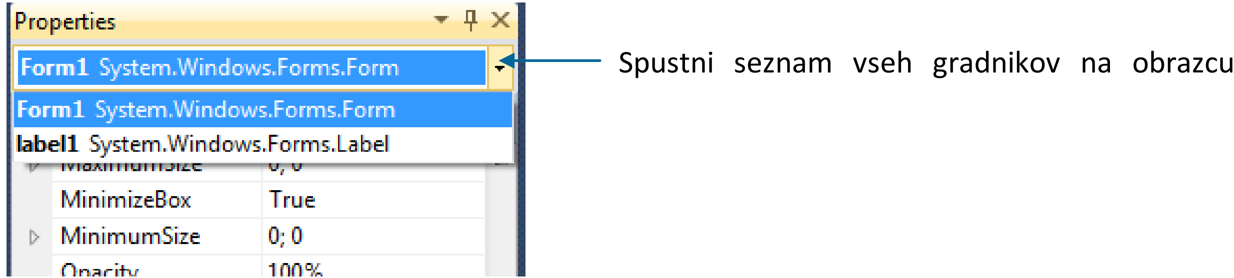
Lastnosti gradnikov v oknu *Properties* so lahko nanizane na dva načina: po abecednem redu ali pa po kategorijah. S klikom izberemo način, ki nam trenutno bolj ustreza.

Prikaz lastnosti po kategorijah

Prikaz lastnosti po abecedi

Slika 7: Urejanje lastnosti gradnikov.

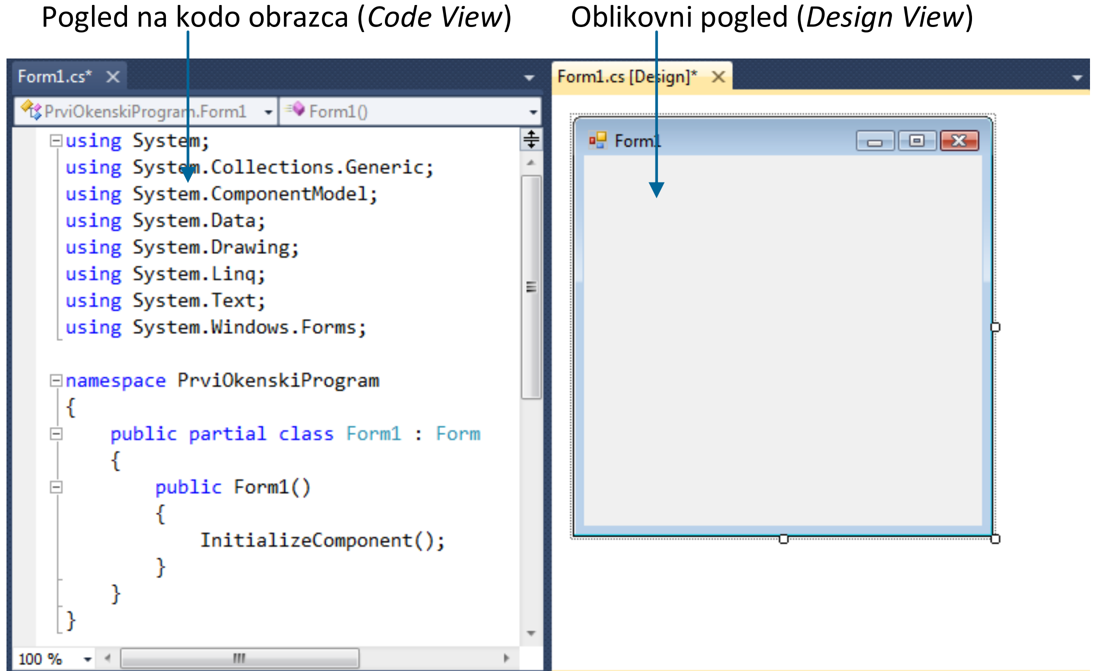
V oknu *Properties* lahko vidimo tudi seznam vseh gradnikov, ki smo jih postavili na obrazec. Poskrbimo za to, da bo izbran gradnik obrazec (*Form1*), nato v oknu *Properties* odpremo spustni seznam vsebovanih gradnikov.



Slika 8: Seznam vseh gradnikov na obrazcu.

Odpre se okno v katerem so nanizani vsi gradniki, ki so trenutno na obrazcu. Izberemo lahko kateregakoli in v oknu *Properties* se prikažejo njegove lastnosti v oknu *Design View* pa je ta gradnik izbran.

Omenjeno je že bilo, da je vsak obrazec v *Visual C#* predstavljen na dva načina: v oblikovnem pogledu (*Form1.cs [design])* in v pogledu za urejanje kode (*Form1.cs*). Oblikovni pogled (*Design View*) in pogled za urejanje kode (*Code View*) sta med seboj neodvisna, tako da lahko na ekranu hkrati prikažemo oba pogleda, kar je še posebej ugodno kadar imamo na razpolago velik monitor.



Slika 9: V urejevalniškem oknu imamo lahko hkrati odprta oba pogleda na obrazec: pogled *Code View* in pogled *Design View*.

Kadar je na vrhu okna (zavihka oz. okna urejevalnika), v katerem je naša koda, znak zvezdica (*), to pomeni, da je bila na tem obrazcu narejena vsaj ena sprememba, potem, ko smo ga nazadnje shranili. Ročno shranjevanje ni potrebno, saj se shranjevanje izvede avtomatsko takoj, ko poženemo ukaz za prevajanje. Seveda pa je ročno shranjevanje priporočljivo takrat, kadar smo zapisali veliko vrstic kode, pa prevajanja še ne želimo pognati.

Slika 10: Zvezdica na vrhu okna pomeni, da zadnje stanje projekta še ni shranjeno.

Oba pogleda v urejevalniškem oknu lahko na ekranu dobimo takole: preklopimo najprej na pogled za urejanje, nato pa z miško primimo zavihek *Form1.cs [design]* in ga povlecimo nad pogled za urejanje kode. Spustimo tipko na miški in prikaže se *Pop Up* meni s tremi opcijami. Izberimo npr. opcijo *New Vertical Tab Group*. Okno za urejanje kode se sedaj razdeli na dva dela: v enem imamo predstavljen pogled za urejanje kode obrazca, v drugem pa oblikovni pogled obrazca. Seveda lahko novo okno kadarkoli zopet zapremo in se tako vrnemo v prejšnji pogled.

Če smo na obrazec pomotoma postavili gradnik, ki ga ne potrebujemo, ga lahko odstranimo na dva načina: označimo ga s klikom miške in pritisnemo tipko *Delete* na tipkovnici, ali pa kliknemo desno tipko miške in v prikazanem *Pop Up* meniju izberemo opcijo *Delete*.

Preglejmo sedaj še vsebino okna **Code View**: Na začetku so stavki, ki predstavljajo vključevanje že obstoječih imenskih prostorov v naš projekt.

```
using System;
```

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

Sledi napoved imenskega prostora, v katerem ustvarjamo naš projekt (njegovo ime je enako imenu projekta) in nato deklaracija razreda *Form1*, ki je izpeljan (deduje – o dedovanju bomo govorili kasneje) iz splošnega razreda *Form* (osnoven, prazen obrazec). Pojasnimo še pomen rezervirane besedice *partial* v glavi razreda *Form1*: ta besedica označuje, da je na tem mestu napisan le del razreda *Form1* (*partial* - delno), ostali del (tisti del, ki ga pri polaganju gradnikov na obrazec za nas gradi razvojno okolje) pa je napisan v drugi datoteki (vsebino te datoteke lahko prikažemo v urejevalniškem oknu tako, da v *Solution Explorerju* kliknemo na vrstico *Form1.Designer.cs*). V konstruktorju razreda Form1 je klicana metoda *InitialComponent*, ki pa se dejansko nahaja v datoteki *Form1.Designer.cs*. To datoteko za nas gradi razvojno okolje. S tem so razvijalci razvojnega okolja poskrbeli za to, da ni celotna koda na enem mestu, kar bi imelo za posledico nepreglednost in težave pri ažuriranju projektov. Bistvena prednost pa je v tem, da je na ta način ločena koda, ki jo pišemo sami in koda, ki jo za nas gradi razvojno okolje. Namesto klica metode *InitialComponent* v konstruktorju, bi v večini primerov lahko na tem mestu zapisali kar celotno vsebino datoteke *Form1.Designer.cs*.

```
namespace PrviOkenskiProgram //imenski prostor, v katerem gradimo naš projekt
{
    //razred Form1 deduje razred Form (osnovni obrazec)
    public partial class Form1 : Form
    {
        public Form1()  //konstruktor razreda Form1
        {
             /*metoda, ki poskrbi za inicializacijo objektov, njihovih
             lastnosti in dogodkov – vsebino te metode si lahko ogledamo
             datoteki Form1.Designer.cs v Solution Explorerju*/
             InitializeComponent();
        }
    }
}
```

Nadaljujmo sedaj z našim prvim projektom. V glavnem oknu kliknemo na ikono *Start Debugging* (ali pa izberemo *Debug→Start Debugging*, ali pa stisnemo tipko *F5*) in naš projekt se bo prevedel (seveda, če je brez napake, a ker doslej še nismo pisali nobene kode, naj napak ne bi bilo). Kot rezultat prevajanja se na zaslonu pojavi novo okno, v katerem se izvaja naš prvi projekt. Zaenkrat je v tem oknu le pozdravno besedilo, ima pa okno v svojem desnem zgornjem kotu tri sistemske ikone za minimiranje, maksimiranje in zapiranje projekta. Obenem je nekje na disku (znotraj mape, ki smo jo na začetku določili pri kliku na *Save All*, in sicer v mapi *Bin→Debug*) nastala datoteka *PrviOkenskiProgram.exe* – to je t.i. izvršilna datoteka (*executable file*). Ta datoteka (program) predstavlja rezultat našega dela (projekta). To je tudi datoteka, ki jo lahko kadarkoli prenesemo na drug računalnik in jo tam izvajamo kot samostojno aplikacijo (seveda je potrebno včasih, zaradi nastavitev, obenem kopirati še kake druge datoteke, a o tem

kasneje). Na ciljnem računalniku pa mora biti seveda nameščeno Microsoftovo okolje (*Framework*).

Okno, v katerem teče naš pravkar prevedeni projekt zaprimo, da se vrnemo nazaj v razvojno okolje. Sedaj lahko zapremo tudi razvojno okolje (ali pa odpremo nov projekt). Še prej s klikom na gumb *Save All* zagotovimo, da je celoten projekt res shranjen!

Če v oknu *Properties* izberemo katerokoli lastnost in nanjo naredimo desni klik, se prikaže okno, v katerem je vrstica *Description*. Če je ta vrstica odkljukana, se nam na dnu okna *Properties* prikaže kratek opis izbrane lastnosti.

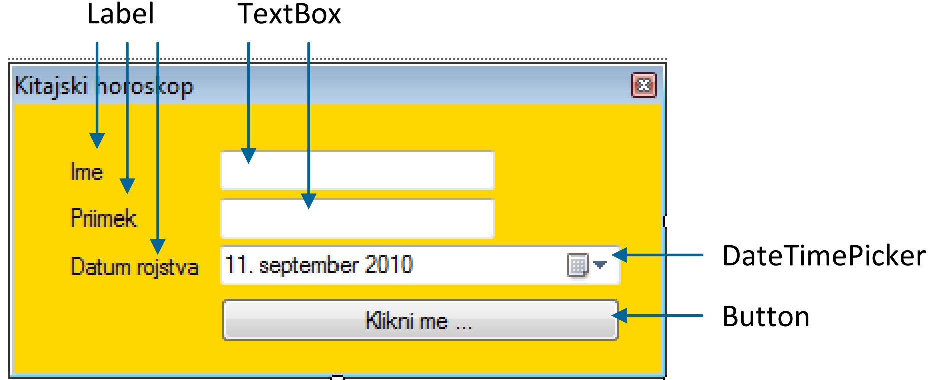
Kitajski horoskop

Radi bi napisali program, ki bo od uporabnika zahteval vnos imena, priimka in rojstnega datuma. Ob kliku na gumb na obrazcu, bo program uporabniku sporočil njegovo znamenje v kitajskem horoskopu!

Ustvarimo torej nov projekt (*File* → *New Project*), ga poimenujmo in shranimo pod imenom *KitajskiHoroskop*. V oknu *Properties* nastavimo obrazcu naslednje lastnosti:

- *Text*: vpišimo besedilo *Kitajski horoskop*
- *FormBorderStyle*: izberimo lastnost *FixedToolWindow*: ta stil okna označuje, da v bo v naslovni vrstici tega okna le en gumb, to je gumb za zapiranje okna. V času izvajanja programa uporabnik okna ne bo mogel minimirati ali maksimirati, niti ga ne bo mogel pomanjšati ali ga povečati.
- *BackColor*: v spustnem seznamu, ki ga lahko odpremo pri tej lastnosti, izberimo jeziček *Web* in v njem npr. barvo *Gold*.

Na tako pripravljen obrazec nato postavimo nekaj gradnikov (vsi se nahajajo na paleti *Common Controls*) in jih razporedimo tako, kot kaže slika:



Slika 11: Gradniki, ki jih potrebujemo na obrazcu *Kitajski horoskop*.

Oglejmo si gradnike, ki smo jih poleg oznake (label) še uporabili. To so:

- *TextBox*: osnovni namen tega gradnika je, da uporabnik programa vanj vnese neko besedilo.
- *DateTimePicker*: gradnik je nemenjen izbiri datuma.
- *Button*: gumb, ki je namenjen izvajanju nekega dogodka, ko uporabnik nanj klikne.

Vsem trem oznakam in gumbu na obrazcu spremenimo lastnost *Text*, da bo besedilo ustrezalo sliki. Lastnosti ostalih gradnikov ne spreminjamo.

Glavni način uporabe gumba (gradnika button) je ta, da napišemo metodo, ki naj se izvede ob uporabnikovem kliku nanj.

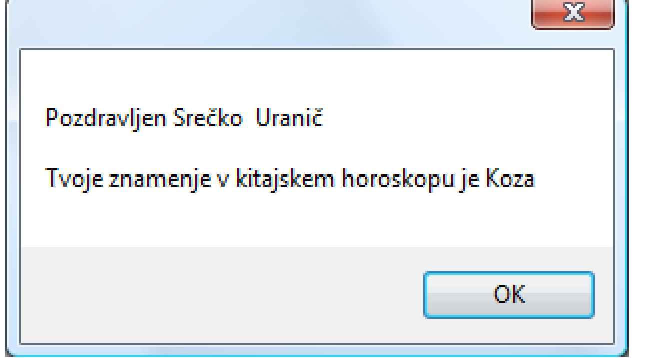
Namreč ena od osnovnih idej pri gradnji okenskih aplikacij je uporaba dogodkovnega programiranja. O tem bomo več povedali v nadaljevanju. Zanekrat povejmo le, da s pomočjo razvojnega okolja za posamezne gradnike napišemo, kaj se bo zgodilo, če pride do določenega dogodka (npr. če klinemo na gumb, če se sopremeni vsebina vnosnega polja …).

Gumb najprej označimo, nato pa v oknu *Properties* kliknemo na ikono *Events* pod vrhom tega okna. Na ta način prikažemo vse že pripravljene dogodke, na katere se gradniki tipa *Button* lahko odzivajo. Imena dogodkov v oknu *Properties* so izbrana tako, da nas asociirajo na njihov pomen. Med dogodki izberemo dogodek *Click* in nanj dvokliknemo. Razvojno okolje nam v datoteki s kodo ustvari ogrodje metode, ki se bo izvedla ob uporabnikovem kliku na gumb. Obenem se v urejevalniškem oknu oblikovni pogled na obrazec spremeni v pogled s kodo obrazca, kurzor pa je že postavljen v telo pravkar ustvarjene metode. Metoda je tipa *void*, njeno ime ustreza namenu. Ima tudi dva parametra, o katerih pa bo več napisanega kasneje. Sedaj moramo le še napisati ustrezne stavke, ki naj se izvedejo ob uporabnikovem kliku na gumb.

Znamenje v kitajskem horoskopu je vezano na leto rojstva. Dobimo ga tako, da preverimo ostanek letnice rojstva pri deljenju z 12: če je le ta enak 0 je znamenje "Opica", če je ostanek 1 je znamenje "Petelin" in tako naprej vse do ostanka 11, ki predstavlja znamenje "Koza".

Potrebovali bomo torej letnico rojstva. Gradnik tipa *DateTimePicker* ima lastnost *Value*. V njej je zapisan celoten datum. Letnico pa nato dobimo s pomočjo lastnosti *Year* torej z *dateTimePicker1.Value.Year*.

Algoritem za določitev znamenja si lahko zamislimo tako, da vsa znamenja shranimo v enodimenzionalno tabelo nizov, indeks v tej tabeli pa predstavlja ostanek pri deljenju letnice z 12, preko katerega pridemo do njegovega imena. Rezultat (niz z znamenjem) prikažemo v sporočilnem oknu *MessageBox*.



Slika 12: Pozdravno sporočilo projekta *Kitajski horoskop*.

Tu ne gre za gradnik, ki bi ga dodali na uporabniški vmesnik, ampak ga ustvarimo v sami kodi. To je v bistvu nek že pripravljen obrazec, namenjen sporočilom uporabniku, prikažemo pa ga z metodo *Show*. Ta metoda ima cel kup različnih načinov uporabe. Na začetku bomo uporabljali le osnovni način: metodi za parameter posredujemo poljuben tekst (niz, ki mora biti seveda v dvojnih narekovajih), ki bo kot sporočilo prikazan v tem oknu. Pri oblikovanju tega teksta se držimo pravil, ki jih že poznamo pri delu z nizi.

Tekst v sporočilnem oknu lahko prikažemo tudi večvrstično tako, da za prehod v novo vrsto uporabimo znakovno konstanto za prehod v novo vrsto (*n* ali pa \n*r*).

```
//celotna koda dogodka, ki se izvede ob kliku na gumb button1
private void button1_Click(object sender, EventArgs e)
{
    //od uporabnika zahtevamo vnos imena in priimka
    if (textBox1.Text.Length == 0 && textBox2.Text.Length == 0)
        MessageBox.Show("Vnesi ime in priimek!");
    else
    {
        //definicija enodimenzionalne tabele znamenj v kitajskem horoskopu
        string[] Kitajski = new string[12] { "Opica", "Petelin", "Pes",
"Merjasec", "Podgana", "Bivol", "Tiger", "Zajec", "Zmaj", "Kača", "Konj",
"Koza" };
        /*Iz izbranega datuma rojstva izluščimo letnico: celoten datum določa
lastnost Value gradnika dateTimePicker1, letnico pa nato dobimo s pomočjo
lastnosti Year*/
        int leto = dateTimePicker1.Value.Year;
        //izračunamo ostanek pri deljenju z 12
        int ostanekPriDeljenjuZ12 = leto % 12;
        /*ostanek pri deljenju z 12 določa indeks znamenja v tabeli  vseh
          znamenj kitajskega horoskopa*/
        string znamenje = Kitajski[ostanekPriDeljenjuZ12];
       //še zaključno obvestilo uporabniku
        MessageBox.Show("Pozdravljen "+textBox1.Text+" " +
textBox2.Text+"\n\nTvoje znamenje v kitajskem horoskopu je " + znamenje);
    }
}
```

Če smo se pri izbiri dogodka zmotili in npr. namesto dogodka Click izbrali na nek drug dogodek, lahko ogrodje metode v datoteki s kodo (v našem projektu je to datoteka *Form1.cs*) odstranimo takole: iz imena dogodka razberemo za kateri gradnik gre (če se npr. ime dogodka prične z *button1*, gre prav gotovo za dogodek gradnika *button1*). Preklopimo na oblikovni pogled in izberemo ustrezen gradnik. V oknu *Properties* nato najprej prikažemo vse dogodke (klik na ikono *Events*) in se pomaknemo v vrstico, kjer vidimo ime dogodka, ki ga želimo odstraniti. Ime (v desnem stolpcu) nato enostavno pobrišemo. Ogrodje dogodka(kodo) v datoteki s kodo bo nato odstranilo razvojno okolje samo, seveda pa mora biti telo te metode prazno: v njej ne sme biti niti komentarja.

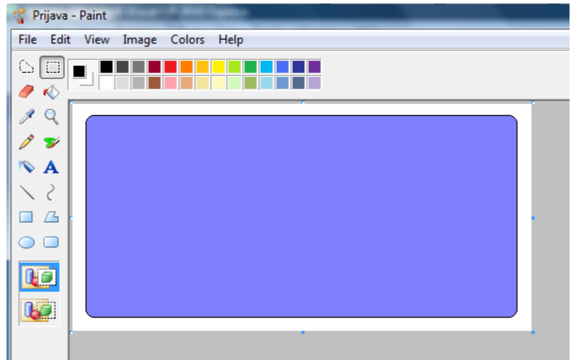
S kakšno metodo se gradnik odzove na določen dogodek povemo tako, da gradnik najprej izberemo, nato pa v oknu *Properties* dvoklinemo v vrstico z ustreznim dogodkom: razvojno okolje bo ob tem pripravilo ogrodje tega dogodka. Vsak

gradnik na obrazcu pa ima enega od dogodkov privzetega. To pomeni, da za ustvarjanje ogrodja tega dogodka ni potrebno le-tega izbrati v oknu *Properties*, ampak obstaja tudi krajši način: na obrazcu le dvokliknemo na ta gradnik in ogrodje odzivne metode je pripravljeno. Privzeti dogodek (to je najbolj značilen dogodek, na katerega se gradnik odziva) gradnika *Button* je *Click*, gradnika *TextBox* dogodek *TextChanged*, itd.

Prijavni obrazec

Slej ko prej bomo želeli napisati projekt, v katerega se bo moral uporabnik preko nekega obrazca najprej prijavit (glej Slika 13).

Privzeto je vsak obrazec pravokotne oblike. Kadar pa želimo ustvariti projekt, v katerem bo obrazec poljubne oblike, lahko to storimo tako, da najprej s pomočjo poljubnega grafičnega urejevalnika najprej pripravimo sliko obrazca (ki mora biti na nekem enobarvnem ozadju). V projektu nato obrazcu to sliko nastavimo kot lastnost *BackgroundImage*, potrebna pa je še nastavitev lastnosti *TansparencyKey*. Ker želimo, da prijavni obrazec ne bo pravokotne oblike, moramo s pomočjo grafičnega urejevalnika (to je lahko kar Slikar) najprej pripraviti ustrezno sliko za ozadje. V našem primeru bomo kar s slikarjem pripravili enobarvno sliko pravokotnika z okroglimi robovi. Da pa bo transparentnost res 100%, sliko shranimo v datoteko tipa *24-bit Bitmap (*.bmp; *.dib)*.



Slika 14: V Slikarju pripravimo ozadje obrazca.

Sliko shranimo in začnimo nov projekt, ki ga poimenujmo *Prijava*. V oknu *Properties* nato obrazcu nastavimo naslednje lastnosti:

- *BackgroundImage*: za ozadje nastavimo pravkar ustvarjeno sliko.
- *BackgroundImageLayout*: *Stretch*(slika naj bo raztegnjena čez celotno ozadje).
- *TransparencyKey*: nastavimo na *White* (ker je pač slika na belem ozadju).
- *FormBorderStyle*: nastavimo na *None*.

Na obrazec nato postavimo naslednje gradnike:

Slika 15: Gradniki na prijavnem obrazcu.

Če se v urejevalniškem oknu *CodeView* z miško postavimo nad neko spremenljivko ali pa objekt, se pod njim pokaže okvirček z besedilom, ki nam pove tip te spremenljivke ali objekta.

Gradnikom na obrazcu pustimo imena kar privzeta, tako, kot jih določi razvojno okolje. Kadar pa bo na obrazcu več podobnih gradnikov, pa je pametno, da gradnike poimenujemo po svoje (*tBIme*, *tBGeslo*, lIme, *lGeslo*, *bZapri*, *bPrijava*, *bNamig* …)

Gradnikoma *TextBox* nastavimo lastnost *Color* (npr. *Gold*, nikakor pa ne belo barvo, ki smo jo uporabili kot ključ za transparentnost) in lastnost *BorderStyle* (*FixedSingle*). Gradniku *TextBox* za vnos gesla nastavimo še lastnost *PasswordChar* (npr. znak zvezdica). Ko bo uporabnik v polje vnašal znake, se bodo na ekranu prikazovale le zvezdice. Spodnji levi gumb na obrazcu je namenjen prikazu sporočilnega okna z namigom o uporabniškem imenu in geslu. Gumbu priredimo še ustrezno sliko (lastnost *BackgroundImage*). Sliko poiščemo v svojem arhivu ali pa s pomočjo spleta, najbolje pa je, da si za naše potrebe vnaprej pripravimo nek nabor slik (*.gif*, *.bmp*, *.jpg, .jpeg*, *.png*, *.wmf*), ki jih bomo potem uvažali v projekte.

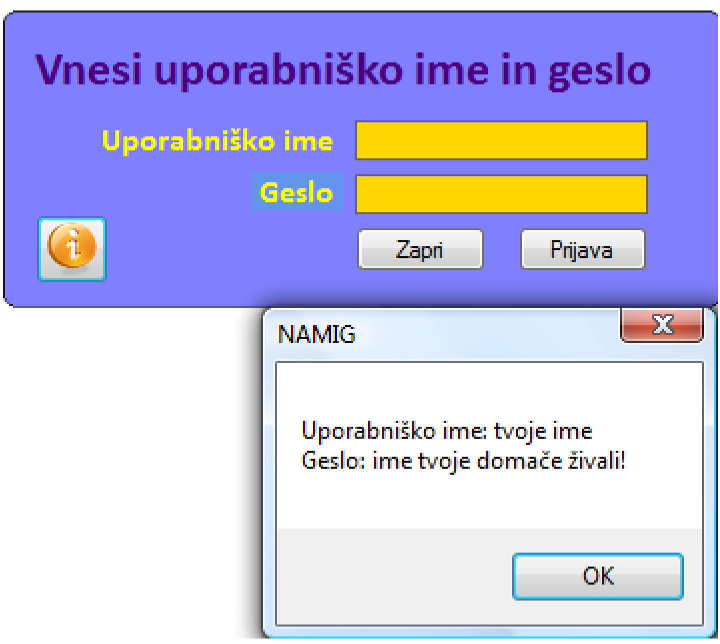
Napisati moramo še kodo za ustrezne dogodke povezane z vsemi tremi gumbi na obrazcu. Tako smo v kodi za klik na gumb *button1* uporabili sporočilno okno s štirimi parametri. Razlaga parametrov je zapisana v komentarju, več o vseh možnih oblikah sporočilnega okna pa zapisano v nadaljevanju.

```
//definicija spremenljivke števila uporabnikovih poskusov vstopa v program
public int poskusov = 0; //uporabniku bomo dovolili le 3 poskuse prijave
private void button1_Click(object sender, EventArgs e)
{
    if (textBox1.Text == "")
        MessageBox.Show("Nisi vnesel uporabniškega imena!");
    else if (textBox2.Text == "")
        MessageBox.Show("Vnesi geslo!");
    //uporabniško ime smo si zamislili kot "Visual C#", geslo pa "prijava"
```

```
    else if ((textBox1.Text == "Visual C#") && (textBox2.Text == "prijava"))
    {
        MessageBox.Show("Geslo pravilno!");//navadno sporočilno okno
        Application.Exit();//zapremo obrazec;
    }
    else
    {
        poskusov++;//povečamo število poskusov
        /*uporabimo sporočilno okno s štirimi parametri: prvi parameter tipa
         niz (string) je izpis v oknu, drugi parameter tipa niz je napis na
         oknu, tretji parameter označuje število in vrsto gumbov, ki se
         prikažejo v oknu, zadnji parameter pa vrsto ikone v tem oknu*/
        MessageBox.Show("Napačno geslo!\n\nŠtevilo dosedanjih poskusov
"+poskusov.ToString()+"\nNa voljo še poskusov: "+(3-poskusov),"Napačno
geslo!",MessageBoxButtons.OK,MessageBoxIcon.Information);
        if (poskusov==3)
            Application.Exit();
    }
}
private void button2_Click(object sender, EventArgs e)
{
    /*v sporočilnem oknu za informacijo izpišemo namig za uporabniško ime in
      geslo. Zopet uporabimo sporočilno okno s štirimi parametri*/
    MessageBox.Show("Uporabniško ime: tvoje ime\nGeslo: ime tvoje domače
živali!","NAMIG", MessageBoxButtons.OK, MessageBoxIcon.Information);
}

private void button3_Click(object sender, EventArgs e)
{
    /*obrazec zapremo z metodo Exit razreda Application, lahko pa tudi kar z
      metodo Close. Razlika med metodama je v tem, da Close zapre trenutno
      okno, Exit pa konča program in zapre še ostala okna.*/
    Application.Exit();
}
```

Seveda je naša aplikacija več ali manj neuporabna, saj ne počne drugega, kot preverja pravilnost gesla. V "pravi" aplikaciji bi seveda po vnosu pravilnega gesla namesto tega, da končamo program, odprli novo okno, ali pa glede na uporabniško geslo odprli točno določen projekt znotraj naše rešitve.

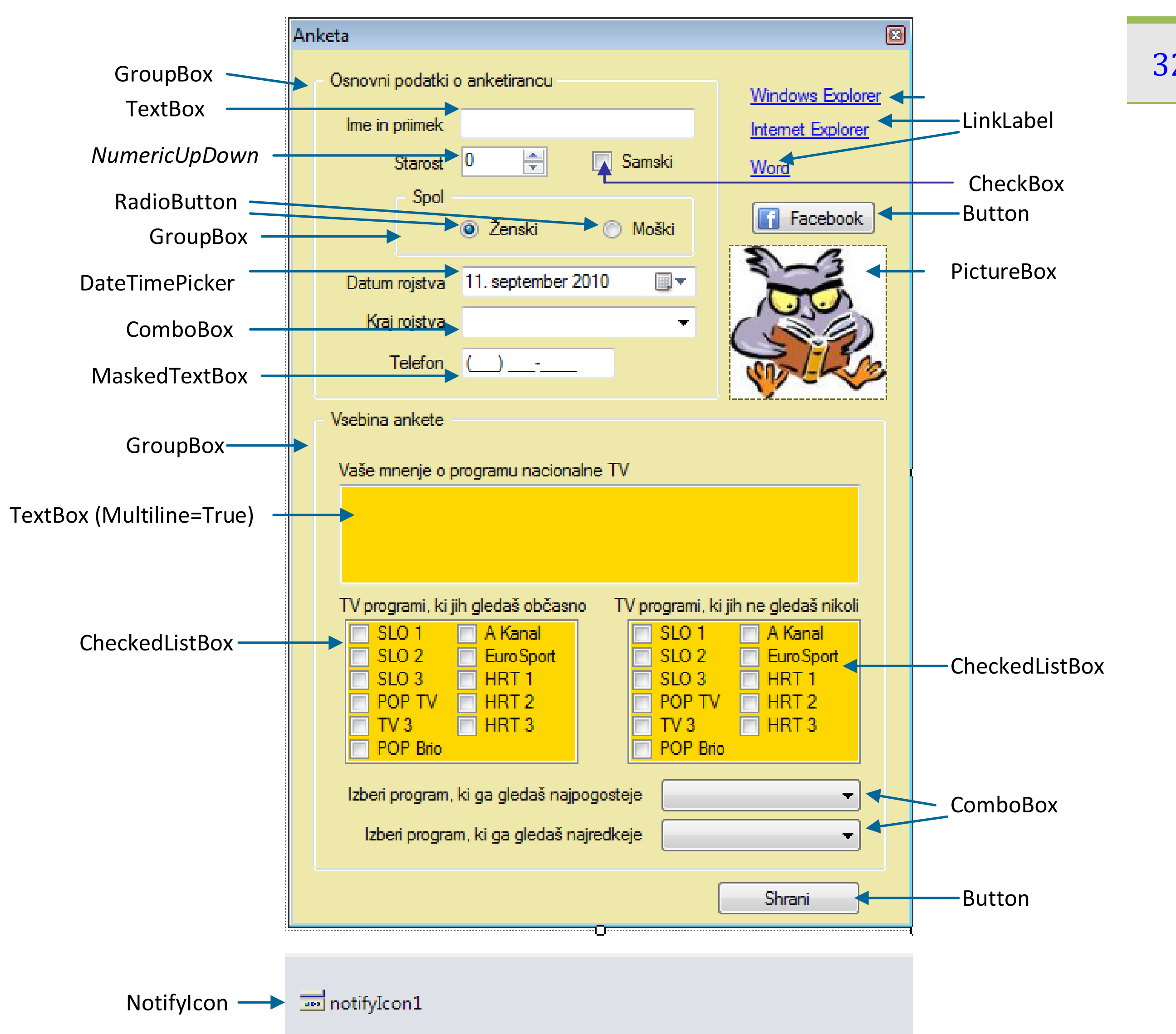
Če projekt zaženemo in kliknemo na spodnji levi gumb, dobimo takole sliko:



Slika 16: Sporočilno okno z nastavitvami na prijavnem obrazcu.

Anketa

Na TV Slovenija so nas prosili, da naj pripravimo obrazec o gledanju TV programov. V ta namen so nam posredovali podatke, ki bi jih želeli izvedeti od TV gledalcev. Rezultate želijo imeti formatirane v ustrezni tekstovni datoteki. Na obrazcu, ki ga bomo za ta namen pripravili, bomo uporabili kopico novih gradnikov: *NumericUpDown*, *CheckBox*, *LinkLabel*, *ComboBox*, *MaskedTextBox*, *PictureBox*, *CheckedListBox*, *NotifyIcon*, *RadioButton* in *GroupBox*. Nekaj o njihovem namenu in najpomembnejših lastnostih si bomo ogledali kar ob izpeljavi tega zgleda. Razporedimo in poimenujmo jih takole:



Slika 17: Gradniki na anketnem obrazcu.

Večina lastnosti gradnikov je privzetih, nekaterim gradnikom pa lastnosti malo popravimo:

- Obrazec *Form1*: lastnost *BackColor* nastavimo na *PaleGoldenrod*, za lastnost *Text* pa zapišimo *Anketa*.
- *LinkLabel*: gradnik ima vlogo povezave (*hyperlink*): njegovemu dogodku *LinkClicked* priredimo zagon poljubnega programa (npr. *Windows Explorer*-ja, *Internet Explorer*-ja, urejevalnika, ..). Gradnikom tega tipa na obrazcu nastavimo lastnost *Text* (tako kot kaže slika) in *VisitedLinkColor* (nastavimo poljubno barvo – ko bo uporabnik kliknil na gradnik, se bo njegova barva ustrezno spremenila).
- *NumericUpDown*: Gradnik je namenjen le vnosu celih števil. Z lastnostma *Max* in *Min* lahko določimo največje in najmanjše število, ki ga uporabnik lahko vnese.
- *GroupBox*: gradnik tega tipa je namenjen združevanju gradnikov, ki po neki logiki spadajo skupaj. V našem primeru ima ta gradnik še poseben pomen pri združevanju radijskih gumbov. Med gumbi, ki so znotraj posameznega obrazca, je lahko naenkrat izbran (ima pikico) le eden. Kadar pa imamo dve (ali več skupin) tovrstnih gumbov,

vsako skupino postavimo znotraj združevalnega gradnika. Pravilo, da lahko uporabnik izbere le en radijski gumb, namreč velja znotraj posameznega združevalnega gradnika. Ko ga postavimo na obrazec, ga lahko premaknemo tako, da ga z miško "primemo" za križec v zgornjem levem robu.

- *ComboBox*: Uporabimo ga za vnos poljubnega števila vrstic besedila, med katerimi bo uporabnik lahko izbiral s pomočjo spustnega seznama. Na obrazcu so trije gradniki tega tipa:
    - *ComboBox* za izbiro kraja: ko ga postavimo na obrazec, najprej kliknemo na trikotnik v desnem zgornjem robu tega gradnika. Odpre se okno, v katerem izberemo opcijo *Edit Items…* (ali pa v oknu *Properties* kliknemo na lastnost *Items*) in vnesemo nekaj krajev (vsakega v svojo vrsto, brez ločil!). Lastnost *DropDownStyle* pustimo nespremenjeno (*DropDown*): na ta način uporabniku omogočimo izbiro enega od že vnesenih krajev, poleg tega pa ima možnost, da v prazno polje vpiše nek drug kraj.
    - Dva tovrstna gradnika za izbiro najbolj in najmanj priljubljenega TV programa. S pomočjo lastnosti *Items…* vnesemo nekaj TV programov, vsakega v svojo vrsto, brez ločil. Obema nastavimo še lastnost *DropDownStyle* na *DropDownList*. Na ta način bomo uporabniku dovolili le izbiro enega od vnesenih programov, ne bo pa mogel vpisati svojega.
- *Button*: zgornjemu gumbu poleg lastnosti *Text* (*Facebook*) določimo še sliko za ozadje.
- *MaskedTextBox*: v gradniku lahko nastavimo ustrezno masko, ki je uporabnikom v pomoč pri vnašanju podatkov, npr. za vnos telefonske številke, datuma, časa, …
- *TextBox*: gradniku za vnos mnenja o programu nacionalne TV nastavimo lastnost *Multiline* na *true*. Na ta način lahko v gradnik zapišemo poljubno število vrstic, lastnost *ScrollBars* pa na *Vertical* (nastavimo navpični drsnik). Lastnost *BackColor* tega gradnika naj bo npr. *Gold*.
- *CheckBox*: gradnik uporabniku omogoča izbiro ali brisanje ponujene možnosti.
- *CheckedListBox*: gradnik tega tipa vsebuje poljubno število gradnikov tipa *CheckBox*. V oba gradnika s pomočjo lastnosti *Items* (klik na tripičje ob lastnosti) vnesemo seznam TV programov, lastnost *BackColor* pa naj bo npr. *Gold*.
- *NotifyIcon*: ta gradnik omogoča prikaz ikonce v vrstici *Windows Taskbar* (običajno na spodnjem desnem robu zaslona). Ko gradnik postavimo na obrazec, se pokaže v prostoru pod obrazcem, na dnu urejevalniškega okna. Gradniku moramo nastaviti ustrezno ikono, za kar uporabimo lastnost *Icon*: kliknemo na tripičje ob lastnosti in s pomočjo okna, ki se odpre, poiščemo ustrezno datoteko tipa *.ico*.

Ostane nam še pisanje kode. Vsem trem gradnikom tipa *LinkLabel* nastavimo odzivne metode *LinkClicked*. Gumbu za zagon *Facebook*-a napišemo odzivno metodo dogodka *Click*, gumbu na dnu obrazca prav tako, v njej pa zapišemo kodo, s pomočjo katere bomo uporabnikove vnose zapisali v tekstovno datoteko *Anketa.txt*. Ta se nahaja v mapi *Bin→Debug* znotraj našega projekta.

Ko pišemo odzivno metodo na določen dogodek pogosto enostavno rečemo, da "nastavimo dogodek".

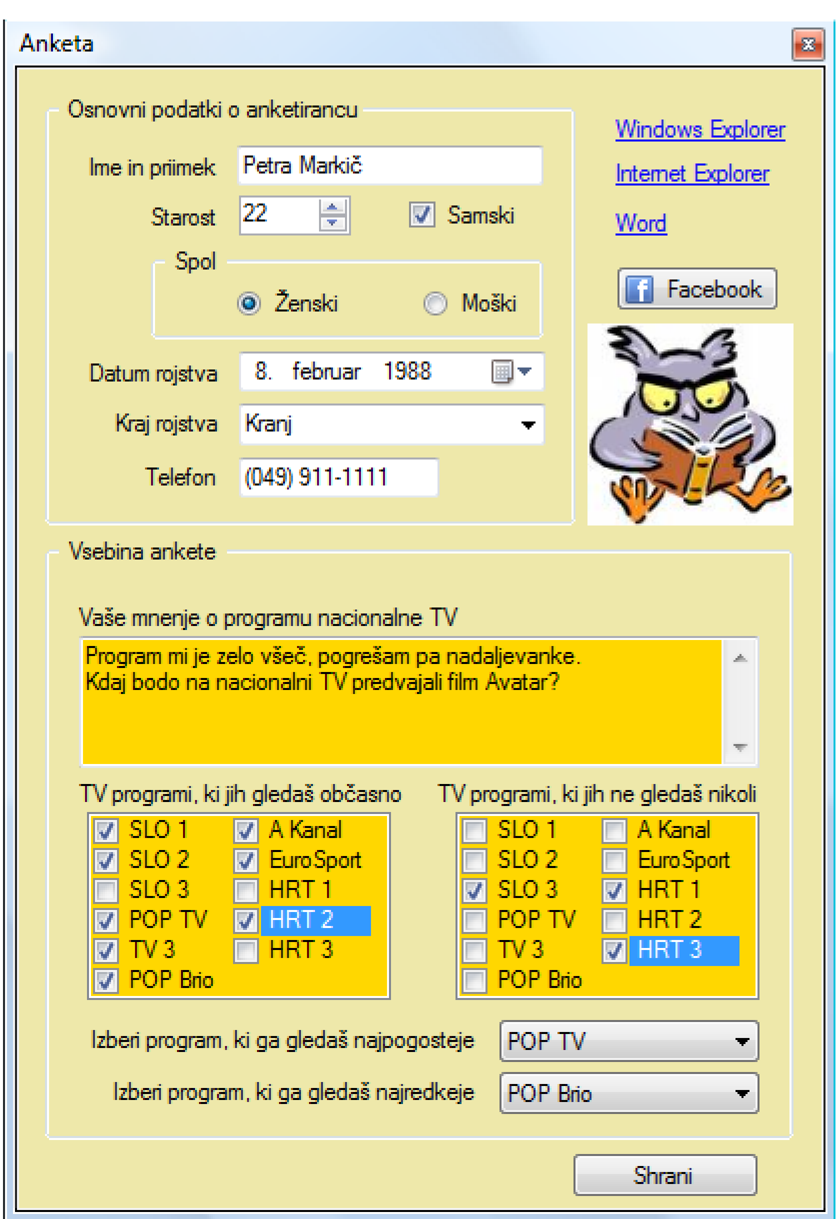
Pri delu se nam bo slej ko prej zgodilo, da bomo v pogledu *Code View* pomotoma pobrisali nek dogodek, ki nam ga je ustvarilo razvojno okolje (bodisi, da smo pomotoma dvokliknili na nek gradnik, ali pa smo v oknu *Properties* dvokliknili na napačen dogodek). Ko projekt nato skušamo prevesti dobimo obvestilo o napaki. Napako odpravimo tako, da v oknu *ErrorList* dvokliknemo na vrstico z napako, nakar nas razvojno okolje postavi v datoteko *.designer.cs* in sicer na mesto napake. Običajno je potrebno vrstico z napako (to je najava nekega dogodka, ki smo ga ravnokar pobrisali) samo pobrisati.

```
//metoda za dostop do Windows Explorerja
private void linkLabel1_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    try  //varovalni blok za obravnavo prekinitev
    {
        /*označimo, da je bila oznaka že kliknena, zato se ji bo barva
          spremenila. Nastavili smo jo z lastnostjo VisitedLinkColor*/
        linkLabel1.LinkVisited = true;
        //metoda Start nas postavi v mapo z rezultati ankete
        System.Diagnostics.Process.Start("C:\\Anketa\\Datum");
    }
    catch
    {
        MessageBox.Show("Napaka pri odpiranju programa ");
    }
}
//metoda za dostop do Internet Explorerja
private void linkLabel2_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    try
    {
        linkLabel1.LinkVisited = true;
        /*metoda Start za zagon Internet Explorer-ja in v njem strani
          S sporedi na RTV SLO1*/
        System.Diagnostics.Process.Start("IExplore",
                                    "http://www.napovednik.com/tv/slo1");
    }
    catch
    {   MessageBox.Show("Napaka pri odpiranju programa ");   }
}
//metoda za zagon urejevalnika Word
private void linkLabel3_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
{
    try
    {
        System.Diagnostics.Process.Start("Winword.exe","Moji zapiski o
oddajah prejšnejega tedna.doc");//zagon Worda in odpiranje datoteke
    }
    catch
    {
        MessageBox.Show("Napaka pri odpiranju programa ");
```

```
    }
}
//metoda za dostop do Facebook-a
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        //metoda Start za odpiranje Facebook - a
        System.Diagnostics.Process.Start("IExplore", "http://sl-
si.facebook.com/");
    }
    catch
    {
        MessageBox.Show("Napaka pri odpiranju programa ");
    }
}
//metoda gumba Shrani za shranjevanje podatkov v tekstovno datoteko
private void button2_Click(object sender, EventArgs e)
{
    /*ime datoteke: ker nismo napisali poti do datoteke, bo le ta shranjena v
      mapi Bin -> Debug tekočega projekta*/
    string datoteka = "Anketa.txt";
    //podatkovni tok za pisanje v tekstovno datoteko
    StreamWriter pisi = File.AppendText(datoteka);
    string ime=textBox1.Text; //shranimo ime in priimek
    //ker je vrednost v gradniku NumericUpDown tipa Decimal, jo z metodo
    //Convert.Toint32 pretvorimo v celo število
    int starost=Convert.ToInt32(numericUpDown1.Value); //starost
    //status je odvisen od izbire oznake v gradniku CheckBox
    string status = "Poročen";
    if (checkBox1.Checked)
        status="Samski";
    string spol="ženski";//spol je odvisen od izbire radijskega gumba
    if (radioButton1.Checked)
        spol="moški";
    string datum=dateTimePicker1.Value.ToShortDateString();//datum
    string kraj=comboBox1.Text;//rojstni kraj
    string telefon=maskedTextBox1.Text; //telefon
    /*najljubše TV programe zapišemo v niz TVProgramiDa*/
    string TVProgramiDa="";
    for (int i = 0; i < checkedListBox1.CheckedItems.Count;i++ )
    {
          //posamezne programe ločimo z vejico
          TVProgramiDa = TVProgramiDa + checkedListBox1.CheckedItems[i]+',';
    }

    /*najmanj priljubljene TV programe zapišemo v niz TVProgramiNe*/
    string TVProgramiNe = "";
    for (int i = 0; i < checkedListBox2.CheckedItems.Count; i++)
    {
         //posamezne programe ločimo z vejico
         TVProgramiNe = TVProgramiNe + checkedListBox2.CheckedItems[i] + ',';
    }
```

```
    string najbolj = comboBox2.Text; //najpogosteje gledani TV program
    string najmanj = comboBox3.Text; //najredkeje gledani TV program
    string komentar = textBox2.Text; //komentar o nacionalni TV
    /*vse podatke zapišemo v datoteko, vmes pa postavimo ločilni znak ; */
    pisi.WriteLine(ime+';'+starost+';'+status+';'+spol+';'+datum+';'+kraj+';'
                   +telefon+';'+TVProgramiDa+';'+TVProgramiNe+';'+najbolj+';'
                   +najmanj+';'+komentar);
    pisi.Close();//zapremo podatkovni tok
    MessageBox.Show("Podatki so shranjeni v datoteki " + datoteka);
    Close();    //zapremo obrazec
}
```

Takole pa je videti obrazec z vnesenimi testnimi podatki:



Slika 18: Anketni obrazec z vnesenimi podatki.

Gradnik *CheckBox* ima lahko tri različna stanja, če je njegova lastnost *ThreeState* nastavljena na *True*. Stanja so *True*, *False* in *Indeterminate* (nedoločeno). Tri stanja so koristna npr. takrat, kadar prikazujemo podatke iz neke relacijske baze. Nekatera polja v bazi podatkov imajo namreč shranjeno vrednost *null*, ki pomeni, da vrednost ni določena ali pa je nepoznana. Kadar želimo to vrednost prikazati tudi v gradniku *CheckBox*, lahko izberemo stanje *Indeterminate*, ki ustreza vrednosti *null* v bazi podatkov.

Samo za vajo si poglejmo še, kako bi podatke, ki so že shranjeni v datoteki *Anketa.txt* prebrali in jih prenesli nazaj na isti obrazec (obraten postopek). Kodo bomo zapisali v metodo, ki se izvede ob dogodku *Load* obrazca. To pomeni, da ko se obrazec prvič odpre (naloži – Load), bodo v njem že podatki iz prve vrstice te datoteke.

```
private void Form1_Load(object sender, EventArgs e)
{
    try
    {
        //datoteko odpremo za branje s pomočjo objekta tipa StremReader
        StreamReader beri = File.OpenText("Anketa.txt");
        //preberemo vrstico in jo shranimo v spremenljivko tipa niz
        string vrstica = beri.ReadLine();
        beri.Close();//datoteko zapremo
        //s pomočjo metode Split dele vrstice, ki so ločeni z znakom podpičje
        //prenesemo v tabelo nizov
        string[] vsebina = vrstica.Split(';');
        textBox1.Text = vsebina[0]; //v celici z indeksom 0 je ime
        //s pomočjo indeksa dostopamo še do ostalih podatkov v tabeli
        numericUpDown1.Value = Convert.ToDecimal(vsebina[1]);
        if (vsebina[2] == "Samski")
            checkBox1.Checked = true;
        if (vsebina[3] == "Ženski")
            radioButton1.Checked = true;
        else radioButton2.Checked = true;
        dateTimePicker1.Value = Convert.ToDateTime(vsebina[4]);
        comboBox1.Text = vsebina[5];
        maskedTextBox1.Text = vsebina[6];
        textBox2.Text = vsebina[11];
        //v celici z indeksom 7 so TV programi, med seboj ločeni z vejico.
        //s pomočjo metode Split jih prenesemo v tabelo daProgrami
        string[] daProgrami = vsebina[7].Split(',');
        foreach (string program in daProgrami)
        {
            for (int i = 0; i < checkedListBox1.Items.Count; i++)
            {
                //pogledamo vsak CheckBox posebej
                if ((string)checkedListBox1.Items[i] == program)
                {
                    checkedListBox1.SetItemChecked(i, true);
                }
            }
        }
        //še programi, ki jih gledamo občasno
        string[] neProgrami = vsebina[8].Split(',');
```

```
        foreach (string program in neProgrami)
        {
            for (int i = 0; i < checkedListBox2.Items.Count; i++)
            {
                //pogledamo vsak CheckBox posebej
                if ((string)checkedListBox2.Items[i] == program)
                {
                    checkedListBox2.SetItemChecked(i, true);
                }
            }
        }
        comboBox2.Text = vsebina[9];
        comboBox3.Text = vsebina[10];
    }
    catch
    {
       MessageBox.Show("Napaka pri branju datoteke!");
    }
}
```

Lastnosti, metode in dogodki gradnikov, razred *object*

Pojasnili smo že, da je koda, s katero se nastavijo lastnosti gradnikov, ki smo jih postavili na obrazec, zapisana v datoteki *.designer.cs*. Tam je poskrbljeno tudi, da se ustvari povezava med dogodkom in gradnikom – torej, da se nastavi s katero metodo naj gradnik reagira ob določenem dogodku. Podatke v tej datoteki navadno nikoli ne spreminjamo sami, saj za vsebino skrbi razvojno okolje samo. Lastnosti spreminjamo tako, da na obrazcu izberemo ustrezen gradnik, nato pa v oknu *Properties* nastavimo/spremenimo poljubno lastnost. Spremembe v datoteki . *designer.cs* bo nato opravilo razvojno okolje samo. Kar nekaj gradnikov smo že spoznali, prav tako njihove osnovne dogodke. Vseh lastnosti, metod in dogodkov gradnikov je preveč, da bi opisovali vsakega posebej, osnovne oz. najpomembnejše med njimi pa bomo spoznavali na konkretnih primerih.

Lastnosti

Pojem lastnost smo se spoznali že pri osnovah objektnega programiranja, pri pisanju lastnih razredov. Vemo, da s pojmom *lastnost* označujemo nastavitev, ki nam pove nekaj o samem gradniku, npr. obrazcu. Lastnosti gradnikom lahko določamo oz. spreminjamo v fazi načrtovanja projekta (v oknu *Properties*), ali pa med samim delovanjem aplikacije s pomočjo programske kode. Doslej smo jih nastavljali oz. spreminjali v glavnem preko okna *Properties*. Lahko pa jih spremenimo tudi v programsko (nekaj takih primerov smo v dosedanjih vajah že imeli). To lahko storimo tako, da v fazi načrtovanja projekta v urejevalniškem oknu (pogled *Code View*) napišemo ime gradnika, zapišemo piko in razvojno okolje nam s pomočjo sistema *IntelliSense* v okenčku pod besedilom prikaže pomoč pri izbiri ustrezne lastnosti oz. metode. To velja za vse gradnike.

Pri programski spremembi poljubne lastnosti obrazca (vendar pa ne vseh, ker so nekatere lastnosti gradnikov *ReadOnly*), se nanj ne moremo sklicevati preko imena (tako kot vedno v razredih), ampak preko parametra *this*, ki smo ga spoznali že pri razredih in objektih. Prevajalnik

namreč vsaki metodi razreda doda še en prameter - kazalec, ki kaže na konkreten objekt (v našem primeru obrazec) nad katerim deluje metoda. Temu parametru je ime *this*. Inicializira se ob klicu metode in ga uporabljamo za dostop do lastnosti tega objekta.

Napis na obrazcu *Form1* bi torej programsko spremenili takole:

```
this.Text = "Tole je spremenjen napis na vrhu obrazca!";
```

Ta stavek lahko zapišemo v katerikoli odzivni dogodek gradnikov, ali pa v nek dogodek obrazca.

Nekateri gradniki (med njimi tudi *TextBox*) imajo, če jih označimo, v desnem zgornjem kotu posebno puščico. Le-ta omogoča dodatne hitre nastavitve, ki pa se razlikujejo od gradnika do gradnika. Pri gradniku *TextBox* je dodatna hitra nastavitev lastnost *Multiline*. Če odkljukamo to možnost, lahko besedilo v gradniku *TextBox* raztegnemo čez večje število vrstic. Kadar želimo besedilo v tem gradniku programsko zapisati v več vrsticah, uporabljamo za prehod v novo vrsto znakovni konstanti \r\n.

Metode

Objekti (gradniki, tudi obrazci) poznajo tudi celo vrsto metod (pravimo jim tudi objektne metode). S pomočjo objektnih metod objektom (tudi obrazec je objekt) povemo, da naj nekaj storijo, oz. da naj se nekaj zgodi. Že pri osnovah OOP pa smo spoznali, da do metod v splošnem dostopamo preko imena objekta, operatorja pika in imenom metode, oziroma preko parametra *this*, npr.:

```
this.BringToFront();//obrazec postavimo pred vse druge obrazce.
```

Najpomembnejše metode obrazca prikazuje naslednja tabela:

Metoda	Razlaga metode
Activate	Lastnost je uporabna predvsem v primeru, ko imamo znotraj projekta odprtih več obrazcev, ki pa ne smejo biti skriti. Obrazec postane aktiven (se postavi v ospredje) in dobi t.i. "fokus".
BringToFront	Obrazec se postavi v ospredje, pred vse ostale obrazce (nasprotje od *SendToBack*).
Close	Zapiranje obrazca, obrazec se uniči.
Hide	Obrazec postane neviden, oz. se skrije. Je pa še vedno v pomnilniku in ga lahko kadarkoli ponovno prikažemo (z metodo *Show* ali *ShowDialog*).
Refresh	Ažuriranje izgleda obrazca in njegova osvežitev (ponoven izris obrazca).
SendToBack	Obrazec se postavi za vse ostale obrazce (nasprotje od *BringToFront*).
SetBounds	Uporablja se za pozicioniranje obrazca.
Show	Prikaz nemodalnega obrazca (normalno okno).

ShowDialog	Prikaz modalnega obrazca (pogovorno okno). Modalno okno je okno, ki ga moramo nujno zapreti, če želimo nadaljevati delo v enem od ostalih oken.

Tabela 2: Najpomembnejše metode obrazca.

Dogodki

Večina gradnikov vizuelnih aplikacij se odziva na dogodke. Dogodki so v bistvu metode, ki pa se zaženejo le tedaj, ko se zgodi nek dogodek sprožen s strani uporabnika, programske kode ali pa sistema. Metodam, ki se prožijo ob neki uporabnikovi akciji (kliku na gradnik, pritisku tipke na tipkovnici, premiku miške …) pravimo tudi *odzivne metode*. Nabor vseh možnih odzivnih metod je zapisan v oknu *Properties→Events* vsakega gradnika posebej. Kot že vemo iz prejšnjih primerov, določeno odzivno metodo sprožimo tako, da se v oknu *Properties* postavimo v prazno polje določenega dogodka, zapišemo ime metode (kadar želimo metodi prirediti svoje ime), ali pa le dvokliknemo v to polje (v tem primeru bo ime metode določilo razvojno okolje: ime bo sestavljeno iz imena gradnika, ki mu odzivna metoda pripada in imena dogodka – npr *button1_Click*).

Ob odprtju vsakega obrazca se najprej zgodi dogodek *Load*. Odzivno metodo za ta dogodek lahko zato uporabimo za programsko nastavljanje lastnosti obrazca. Pogosto vanj zapišemo tudi začetne vrednosti spremenljivk in objektov. Tega dogodka ne smemo zamenjati z dogodkom *Activated*, saj je med njima velika razlika. Dogodek *Load* se izvede le enkrat, ko so obrazec odpira prvič, dogodek *Activated* pa najprej na samem začetku, takoj za dogodkom *Load*, nato pa vselej ko se obrazec ponovno prikaže (če ga npr. minimiziramo, pa potem zopet prikažemo).

Najpomembnejše dogodke obrazca prikazuje naslednja tabela:

Dogodek	Razlaga, kdaj se dogodek zgodi
Load	Ob nastanku obrazca, ko je obrazec prikazan prvič.
Activated	Ko postane obrazec (ki ga je uporabnik skril npr. z metodo Hide) ponovno aktiven.
Deactivate	Ko aktiven obrazec postane neaktiven (npr. ko se uporabnik premakne na drug obrazec s klikom miške ali pa npr. preko neke bližnjice LinkLabel na neko spletno stran, ali pa je npr. lastnost ActiveForm programsko prirejena drugemu obrazcu).
Shown	Dogodek se zgodi, ko je obrazec prikazan prvič. V primeru, da je bil obrazec šele ustvarjen, se najprej zgodi dogodek *Load*, nato pa šele dogodek *Shown*. Dogodek pa se ne zgodi npr. ob maksimiranju obrazca, ali pa ob ponovnem izrisu obrazca in podobno.
FormClosing	Ob zapiranju obrazca, preden se obrazec zapre (z ukazom *Close* ali gumbom za zapiranje). Preden se obrazec zapre, se sprostijo oz. uničijo vse spremenljivke in objekti, ki so bili deklarirani na obrazcu (oz. na splošno v

	gradniku, ki se zapira).
FormClosed	Ko se obrazec že zapre (z ukazom *Close* ali gumbom za zapiranje).
Paint	Ob ponovnem izrisu obrazca (npr. tedaj, ko je bil obrazec delno zakrit ali pa pomanjšan).
Resize	Ob spremembi velikosti obrazca.

Tabela 3: Najpomembnejši dogodki obrazca.

Preimenovanje obrazcev in ostalih gradnikov in njihovih dogodkov

Včasih se zgodi, da želimo že pripravljen obrazec preimenovati. To lahko storimo na dva načina:

- V oknu *Solution Explorer* izberemo obrazec, ki ga želimo preimenovati (npr. *Form1.cs*) in ga preimenujemo: najprej ga izberemo, kliknemo desni miškin gumb in nato izberemo opcijo *Rename*. Seveda pa ga lahko preimenujemo tudi tako, da v oknu *Solution Explorer* ime obrazca najprej izberemo (ga kliknemo), počakamo nekaj trenutkov, nato pa ga še enkrat kliknemo. Po nekaj trenutkih se prejšnje ime obda z okvirčkom. Nato kar v njem spremenimo ime obrazca (končnice *cs* ni potrebno pisati, doda se avtomatsko). Po spremembi imena nam razvojno okolje v sporočilnem oknu izpiše obvestilo, da smo spremenili ime datoteke in nas vpraša, če želimo preimenovati tudi vse reference, ki se v tem projektu nanašajo na ta obrazec. Izberemo gumb *Da*!

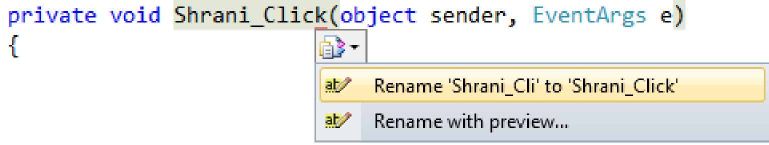
V oknu *Properties* izberemo lastnost *Name* tega obrazca in spremenimo ime. A v tem primeru bomo spremenili le ime temu obrazcu, ne pa tudi reference na ime. Ime obrazca v oknu *Solution Explorer* bo ostalo nespremenjeno. Boljši način za preimenovanja obraza je torej prvi.

Kadarkoli lahko preimenujemo tudi imena gradnikov in odzivnih metod na katere se ti gradniki odzivajo!

Ime gradniku lahko spremenimo tako, da ga najprej izberemo, nato pa v oknu *Properties* spremenimo nastavitev (*Name*) tega gradnika. Priporočljivo je, da gradnikom (pa tudi dogodkom) prirejamo zveneče imena, kar pomeni, da naj nas ime asociira na pomen gradnika. Tako bomo npr. gradniku *TextBox*, ki ga bomo uporabljali za vnos imena dali ime *tBIme* (ali pa npr. *vPIme*) gumbu za shranjevanje naj bo ime *bShrani(ali pa bShrani)*, ipd. Pri poimenovanju gradnikov smo uporabili t.i. *camelCase* konotacijo. Ob spremembi imena bo razvojno okolje avtomatično spremenilo tudi imena vseh referenc (kjerkoli v kodi) na ta gradnik.

- Imena odzivnih metod pa lahko spremenimo neposredno tudi v kodi. Recimo, da bi želeli ime odzivne metode *button1_Click* spremeniti v *Shrani_Click*. Postavimo se v glavo metode, ki ji želimo spremeniti ime in ime spremenimo. Po spremembi imena se pod zadnjim znakom novega imena metode pojavi majhen kvadratek: nanj se postavimo z miško, odpremo prikazan spustni seznam, v njem pa izberemo opcijo *Rename button1_Click to Shrani_Click (*a to moramo storiti takoj, neposredno po spremembi imena, sicer nam razvojno okolje pri tem ne bo več v pomoč!). Vse potrebne spremembe (tudi reference na ta dogodek) bo za nas nato opravilo razvojno okolje.

Poleg konotacije imenovane *camelCase* poznamo še konotacijo imenovano *PascalCase*. Edina razlika med njima je ta, da se imena pri prvi začenjajo z malo črko, pri drugi pa z veliko črko. *Microsoft* priporoča, da za imena spremenljivk uporabljamo konotacijo *camelCase*, za imena metod pa konotacijo *PascalCase*.



Slika 19: Sprememba imena odzivnega dogodka.

Odzivne metode lahko poimenujemo po svoje že pri samem ustvarjanju. Če želimo npr. ustvariti odzivno metodo dogodka *Click* nekega gumba, se v oknu *Properties→Events* najprej postavimo v polje *Click*. Namesto da v to polje dvokliknemo, zapišemo najprej poljubno ime metode (npr *KlikGumba*), ter šele nato dvokliknemo v to polje. Razvojno okolje bo za ime odzivne metode uporabilo naše ime.

Vse odzivne metode obrazca in gradnikov so tipa *void*: nič ne vračajo, le nekaj naredijo. Običajno so zasebne (*private*), kar pomeni da so dostopne le znotraj trenutnega razreda (obrazca, ki ga gradimo).

Parametri odzivnih metod (dogodkov).

Napišimo še nekaj o parametrih odzivnih metod:

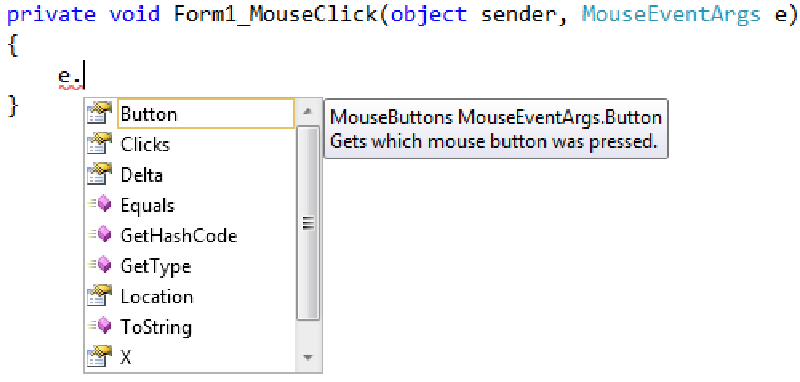
- prvi parameter vsake odzivne metode je parameter *sender*, ki je tipa *object*. Vsem razredom v C#, ne glede na vrsto, je skupno, da je njihov skupni osnovni razred *System.Object*. Sem so všteti tudi osnovni in vsi ostali vrednostni tipi (tudi *int*, *float*, *decimal*, …). Vsi razredi imajo torej avtomatično za svojo osnovo (pravimu tudi, da *dedujejo* - več o dedovanju bo napisano v posebnem poglavju) razred *Object*. To pomeni, da avtomatsko že imajo na voljo metode in lastnosti, ki so napisane v tem razredu. Prav tako velja, da so vsi objekti torej sočasno tudi tipa *object*. Vse to so razvijalci jezika s pridom izkoristili tudi pri odzivnih metodah gradnikov. Ne glede na to, preko katerega gradnika (to je objekta izpeljanega iz nekega razreda, npr. objekta *button1* izpeljanega iz razreda *Button*, objekta *textBox1* izpeljanega iz razreda *TextBox*, ipd.) smo v oknu *Solution Explorer* (ali pa z dvoklikom na gradnik) ustvarili ogrodje neke metode, vedno je njen prvi parameter imenovan *sender* (pošiljatelj), ki je splošnega tipa *object*. S pomočjo tega parametra lahko ugotovimo, nad katerim gradnikom se je zgodil določen dogodek. Poznavanje parametra *sender* je nepogrešljivo npr. tedaj, ko je na obrazcu veliko število gumbov in nam ni potrebno pisati odzivnih metod za vsak gumb posebej: enemu od njih napišemo odzivno metodo, ostalim pa v oknu *Properties→Events* ta isti dogodek le priredimo. Znotraj metode pa potem npr. s pomočjo stavka *if* preverjamo ime gradnikov in zapišemo ustrezno kodo.

  Razred *object* kot takega (samostojno) uporabljamo le redko, a njegovo poznavanje je nepogrešlivo. Ima le privzeti konstruktor, pozna pa le nekaj metod. Omenimo le tri:

Metoda	Razlaga
Equals	Oceni, ali sta dva objekta enaka ali ne.
GetType	Omogoča dostop do tipa nekega objekta.
ToString()	Pretvorba objekta v niz - *string*.

Tabela 4: Metode razreda *object*.

- Drugi parameter odzivnih metod je dogodkovni parameter (objekt tipa *EventArgs*, ali pa nekega izpeljanega razreda – pojem izpeljanega razreda bo pojasnjen v poglavju o dedovanju) imenovan *e*. To je objekt, ki pogosto vsebuje številne lastnosti s pomočjo katerih lahko pridemo do informacij o samem dogodku, v katerem se nahajamo. Do seznama teh lastnosti najlaže pridemo tako, da v telesu odzivne metode nekega dogodka zapišemo oznako parametra, za njim pa piko, nakar nam sistem *IntelliSense* v okvirčku izpiše seznam vseh možnih lastnosti. Pri argumentih tipa *EventArgs* je teh lastnosti malo, pri nekaterih izpeljanih razredih pa precej več. Pri izpeljanem razredu *MouseEventArgs* (dogodku miške) lahko npr ugotovimo, kateri gumb je bil pritisnjen. Več o dogodkih miške pa bo zapisanega v nadaljevanju.



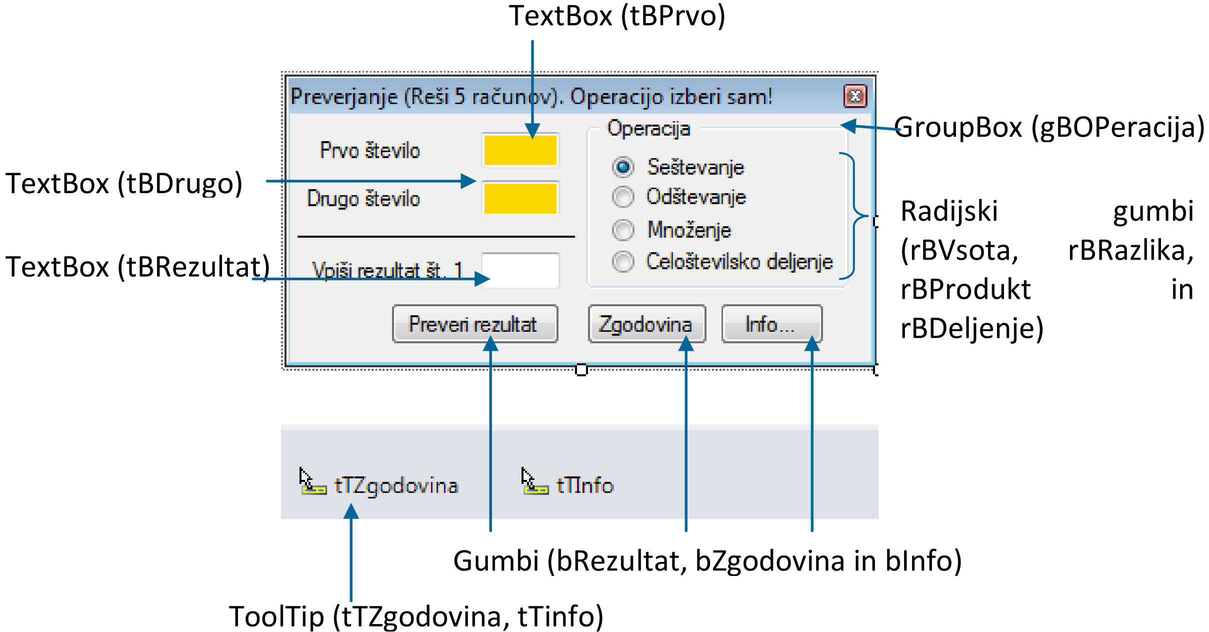
Slika 20: Lastnosti drugega parametra odzivnih metod.

S pomočjo parametra sender lahko preverimo (ali pa določimo) tudi tip objekta (ali pa gradnika, ki nas je poslal v določeno metodo). To lahko storimo s pomočjo operatorjev *is* in *as*, ki bosta razložena v nadaljevanju!

Preverjanje znanja osnovnih računskih operacij

Učitelj matematike v osnovni šoli nas je prosil, naj napišemo program za preverjanje znanja osnovnih računskih operacij. Učenci naj imajo na voljo 5 računov, operacijo (seštevanje, odštevanje, množenje in celoštevilsko deljenje) pa naj izberejo sami. Števila naj bodo naključna, vrednosti pa med 1 in 10. Število pravilnih odgovorov v vsakem poskusu naj se zapisuje v tekstovno datoteko, zraven pa še datum in ura poskusa.

Vse gradnike na obrazcu bomo poimenovali po svoje. Obrazec smo preimenovali *fPreverjanje*, imena ostalih gradnikov pa so razvidna s slike. Gradniki, ki niso poimenovani posebej, so vsi tipa *Label* in imajo privzeta imena - tudi vodoravna črta pod napisom "Drugo število" je tipa label. Njenalastnost *Text* je enaka "__________". Uporabili smo tudi gradnika *ToolTip*. Namen tega gradnika je prikaz pomoči (kratke informacije) v oblačku, če se, (ko je projekt zagnan) z miško postavimo nad gradnik, ki smo mu ta *ToolTip* priredili. V našem primeru smo gradnika vrste *ToolTip* priredili gumboma z napisom *Zgodovina* in *Info*.



Slika 21: Gradniki in njihova imena na obrazcu *FPreverjanje*.

Lastnosti gradnikov na obrazcu so prikazane v naslednji tabeli:

Gradnik	Lastnost	Nastavitev	Opis
Fpreverjanje	FormBorderStyle	FixedToolWindow	Uporabnik velikosti okna ne more spreminjati
tBPrvo	BackColor	Gold	
	TabStop	False	
	TextAlign	Right	Tekst v oknu bo poravnan desno
tBPrvo	BackColor	Gold	
	TabStop	False	
	TextAlign	Right	Tekst v oknu bo poravnan desno
rBVsota	Checked	True	Gumb je na začetku označen

bZgodovina	ToolTip on tTZgodovina	Dosedanji računi	Komentar v oblačku
bInfo	ToolTip on tTInfo	Informacije o programu	Komentar v oblačku
tTZgodovina	IsBaloon	True	Obvestilo bo v oblačku
	ToolTipIcon	Info	V oblačku bo ikona Pomoč
	ToolTipTitle	Pomen gumba	Zapis na vrhu oblačka
tTInfo	IsBaloon	True	Obvestilo bo v oblačku
	ToolTipIcon	Info	V oblačku bo ikona Pomoč
	ToolTipTitle	Pomen gumba	Zapis na vrhu oblačka

Tabela 5: Lastnosti gradnikov na obrazcu z imenom *FPreverjanje*.

V projektu smo uporabili tudi odzivno metodo *Load* obrazca: v njem smo poskrbeli za inicializacijo spremenljivk, ter za klic lastne metode *ZacetnaVrednost*, ki generira naključni števili in jih zapiše v gradnika *tBPrvo* in *tBDrugo*. Gumboma *bZgodovina* in *bInfo* smo priredili isto odzivno metodo z imenom *Info* (ime dogodka smo določili sami). Znotraj dogodka (metode) *Info* s pomočjo parametra *sender* ugotavljamo, kateri od gumbov je bil pritisnjen. Ob kliku na gumb z napisom *Zgodovina* se nam v sporočilnem oknu prikaže vsebina datoteke *Rezultati.txt* (dosedanji rezultati), ob kliku na gumb z napisom *Info* pa informacije o programu! Tudi radijski gumbi krmilijo skupni dogodek z imenom *rBVsota_CheckedChanged* (dogodek se izvede, če spremenimo izbiro radijskega gumba).

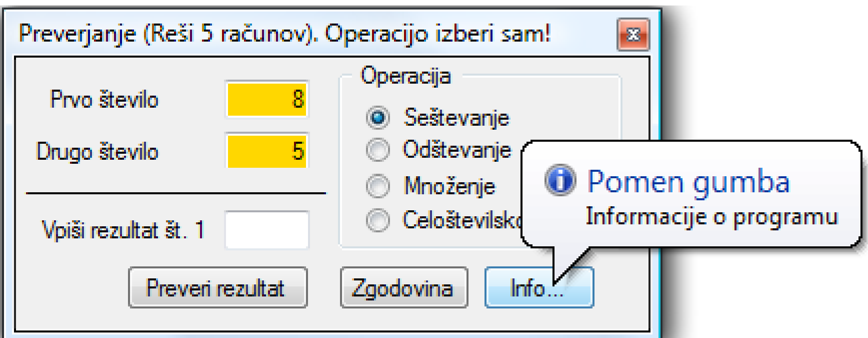
Dogodki, ki so prirejeni posameznim gradnikom so opisani v spodnji kodi obrazca:

```
public partial class FPreverjanje : Form
{
    public FPreverjanje()//konstruktor (privzeti)
    {
        InitializeComponent();
    }
    //javne spremenljivke/objekti
    public int stevilo1, stevilo2, rezultat,stevec,pravilnih;
    Random naklj = new Random();
    //zapomnimo si ime datoteke, saj ga bomo potrebovali večkrat
    public string imeDat = "Rezultati.txt";
    //dogodek Load se zgodi preden se obrazec prvič odpre
    private void FPreverjanje_Load(object sender, EventArgs e)
    {
        //števec vseh in števec pravilnih odgovorov
        stevec = 0; pravilnih = 0;
        //klic metode za generiranje dveh naključnih števil
        ZacetnaVrednost();
    }
    //zasebna metoda za generiranje dveh naključnih števil, ki ju vpišemo v
    //gradnika textBox1 in textBox2
```

```
    private void ZacetnaVrednost()
    {
        //ustvarimo dve naključni celi števili med 1 in 10
        stevilo1 = naklj.Next(1, 11);
        stevilo2 = naklj.Next(1, 11);
        //obe števili zapišimo v gradnika textBox1 in taxtBox2
        tBPrvo.Text = stevilo1.ToString();
        tBDrugo.Text = stevilo2.ToString();
        tbRezultat.Clear();//pobrišemo vsebino gradnika textBox3
        //želimo, da gradnik textBox3 postane aktiven gradnik na našem
        //obrazcu – ima fokus
        tbRezultat.Focus();
        stevec++;//povečamo števec računov
        label3.Text = "Vpiši rezultat št. " + stevec.ToString();
    }
    private void bRezultat_Click(object sender, EventArgs e)
    {
        try
        {
            /*v spremenljivko rezultat shranimo uporabnikov odgovor, ki je v
              textBox3*/
            rezultat = Convert.ToInt32(tbRezultat.Text);
            bool OK = false;
            if (rBVsota.Checked)
             /*spremenljivki OK priredimo vrednost true ali false, glede na
              pravilnost oz. nepravilnost primerjanja :
              rezultat == stevilo1 + stevilo2*/
                OK = (rezultat == stevilo1 + stevilo2);
            /*lahko bi napisali tudi  if (rezultat == stevilo1 + stevilo2)
                                         OK = true;*/
            else if (rBRazlika.Checked)
                OK = (rezultat == stevilo1 - stevilo2);
            else if (rBProdukt.Checked)
                OK = (rezultat == stevilo1 * stevilo2);
            else if (rBDeljenje.Checked)
                OK = (rezultat == stevilo1 / stevilo2);
            //preverimo, če je rezultat OK
            if (OK)
            {
                MessageBox.Show("Odgovor JE pravilen!");
                pravilnih++; //povečamo števec pravilnih odgovorov
            }
            else
                MessageBox.Show("Odgovor NI pravilen!");
            if (stevec == 5)
            {
                string ocena = "";
                switch (pravilnih) //preverimo število pravilnih odgovorov
                {
                    case 0: ocena = "Nezadostno"; break;
                    case 1: ocena = "Nezadostno"; break;
                    case 2: ocena = "Zadostno"; break;
                    case 3: ocena = "Dobro"; break;
```

```
                case 4: ocena = "Prav dobro"; break;
                default: ocena = "Odlično"; break;
            }
            //Izpis rezultata
            MessageBox.Show("Pravilnih odgovorov: "
                + pravilnih.ToString()
                + "\nNepravilnih odgovorov: "
                + (stevec - pravilnih).ToString()
                +"\n\nOcena: "+ocena);
            /*rezultat shranimo v tekstovno datoteko Rezultati.txt. Ob
            vsakem rezultatu naj bo zapisan tudi datum in ura reševanja*/
            //metoda AppendText odpre datoteko za dodajanje besedila. Če
            //datoteka še ne obstaja, jo ustvari, datoteka pa ostane
            //odprta za pisanje.
            StreamWriter pisi = File.AppendText(imeDat);
            //rezultat zapišemo v datoteko
            pisi.WriteLine("Datum: " + DateTime.Now.ToShortDateString() +
", Ura: " + DateTime.Now.ToShortTimeString() + ", Pravilnih odgovorov: " +
pravilnih.ToString());
            pisi.Close();//zapremo podatkovni tok
            /*števec vseh odgovorov in števec pravilnih odgovorov
              postavimo na 0, na vrsti je novih 5 računov*/
            stevec = 0; pravilnih = 0;
        }
        //klic metode za generiranje novih dveh naključnih števil
        ZacetnaVrednost();
    }
    catch
    {
        MessageBox.Show("Napaka v podatkih!");
    }
}
//odzivna metoda, ki se izvede ob kliku na gradnik gBOperacija
private void gBOperacija_Enter(object sender, EventArgs e)
{
    tbRezultat.Focus();
}
/*Info je odzivna metoda dogodka Click gumbov bZgodovina in bInfo.
  Poimenovali smo jo po svoje!*/
private void Info(object sender, EventArgs e)
{
    if (sender == bZgodovina)/*če je bil kliknjen gumb z napisom
                                Zgodovina*/
    {
        //ime datoteke z dosedanjimi rezultati
        //uporabimo varovalni blok za obdelavo prekinitev
        try
        {
            //celotno vsebino datoteke preberemo naenkrat
            string vsebina=File.ReadAllText(imeDat);
            //in jo izpišemo v sporočilnem oknu
            MessageBox.Show(vsebina);
        }
```

```
            catch //v primeru napake naj se izpiše le ustrezno besedilo
            { MessageBox.Show("Napaka pri obdelavi datoteke " + imeDat); }
        }
        else if (sender==bInfo)/*če je bil kliknjen gumb z napisom Info...*/
        {
            //pripravimo informacijo za uporabnika
            string info="V polje 'Vpiši rezultat št. ' vtipkaj rezultat in 
klikni gumb 'Preveri rezultat'.";
            //dodajamo nove vrstice
            info+="\nPo petih računih se bo izpisalo število pravilnih 
odgovorov in ocena!";
            info+="\nDatum, ura poskusa in rezultati se obenem zapisujejo v 
tekstovno datoteko 'Rezultati.txt'.";
            info+="\n\nVsebino te datoteke dobiš s klikom na gumb 
'Zgodovina'.";
            //informacije izpišemo v sporočilnem oknu
            MessageBox.Show(info);
        }
    }
    /*če kliknemo na katerega koli od radijskih gumbov, se vsebina polja 
      tbRezultat briše*/
    private void rBVsota_CheckedChanged(object sender, EventArgs e)
    {
        tbRezultat.Clear();
    }
}
```



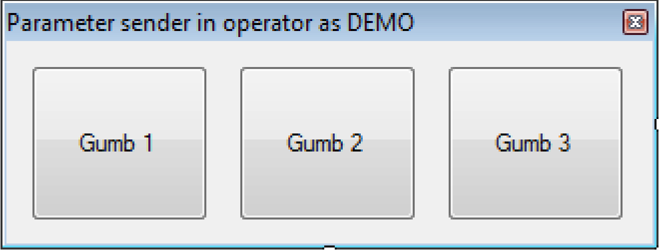
Slika 22: Če se v delujočem programu z miško postavimo na gumb *Info*, se nam prikaže oblaček z ustrezno informacijo!

Operatorja *is* in *as*

S pomočjo parametra *sender* lahko tudi preverjamo tip objekta (gradnika), dostopamo in tudi spreminjamo lastnosti gradnikom, oziroma poganjamo njihove metode. Pomagamo si z operatorjema ***is*** in ***as***.

Parameter *as* omogoča dostop do lastnosti poljubnega objekta ali pa gradnika (preko imena gradnika ali pa preko parametra *sender*). Recimo, da imamo na obrazcu tri gumbe (imena pustimo privzeta: *button1*, *button2* in *button3*). Vsi trije gumbi naj krmilijo isto odzivno metodo *Click*, ki smo jo poimenovali *Gumb_Click* (vsem trem gumbom v oknu *Properties→Events*

priredimo/določimo isti odzivno metodo). Ob kliku na kateregakoli od teh treh gumbov želimo spremeniti nekaj njegovih lastnosti, kar dosežemo s pomočjo parametra *sender* in operatorja *as*.

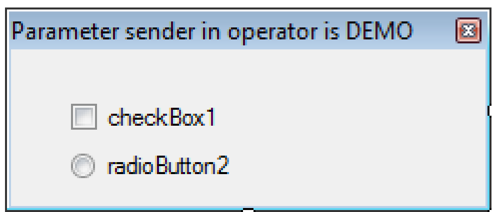


Slika 23: Parameter *sender* in operator *as*.

```
private void Gumb_Click(object sender, EventArgs e)
{    //spremenimo napis na gumbu, ki je bil kliknjen
    (sender as Button).Text = "Skrit";
    //kliknjen gumb naj postane neaktiven (ne moremo ga več "klikniti")
    (sender as Button).Enabled = false;
}
```

Lastnosti izbranega gradnika smo torej spremenili brez preverjanje imena gradnika.

S pomočjo parametra *is* pa lahko preverimo tip poljubnega objekta ali pa gradnika (preko imena gradnika, ali pa preko parametra *sender*). Za primer postavimo na obrazec dva gradnika: gradnik *CheckBox* in gradnik *RadioButton* (imeni gradnikov naj bosta privzeti). Obema gradnikoma priredimo isto odzivno metodo dogodka *ChechedChanged*, ki jo poimenujemo *Sprememba*. S pomočjo parametra *sender*, ter operatorjev *is* in *as* lahko kontroliramo lastnosti obeh gradnikov v isti odzivni metodi.



Slika 24: Parameter *sender* in operator *is*.

```
private void Sprememba(object sender, EventArgs e)
{
    if (sender is CheckBox)//preverimo, če je bil kliknjen gradnik CheckBox

    {
       if ((sender as CheckBox).Checked)//če je izbran
         (sender as CheckBox).Text="Kontrolnik izbran";//napis ob kontrolniku
       else
         (sender as CheckBox).Text="Kontrolnik NI izbran";
    }
    //preverimo, če je bil kliknjen radijski gumb
```

```
    if (sender is RadioButton)
    {
        if ((sender as RadioButton).Checked) //če je izbran
            (sender as RadioButton).Text = "Gumb JE izbran";
    }
}
```

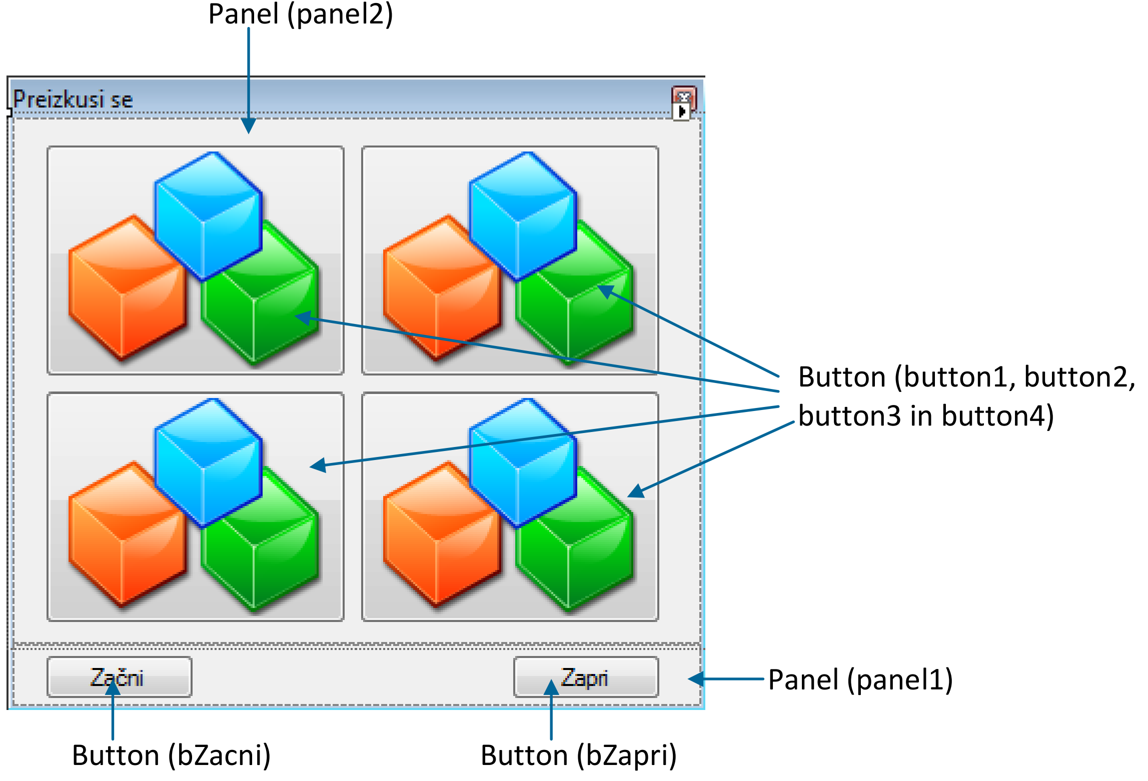
Hitrostno klikanje

Radi bi se preizkusili v hitrostnem klikanju. V ta namen ustvarimo obrazec s štirimi gumbi. Klikniti je možno le na gumb, ki je aktiven. Kateri gumb je aktiven določimo programsko (naključno). Merimo čas od začetka do takrat, ko bomo 20x zaporedoma pravilno kliknili na aktivni gumb. Če gumb zgrešimo, se torej števec zadetkov postavi nazaj na 0.

Lastnosti gradnikov na obrazcu so prikazane v naslednji tabeli:

Gradnik	Lastnost	Nastavitev	Opis
FIgra	FormBorderStyle	FixedToolWindow	Uporabnik velikosti okna ne more spreminjati.
FIgra	StartPosition	Center	Projekt se odpre na sredini zaslona.
panel1	Dock	Bottom	Gradnik je "prilepljen" na dno obrazca.
panel2	Dock	Fill	Gradnik je razširjen nad celotni preostali del obrazca.
button1, button2, button3, button4	Image	Poljubna slika	

Tabela 6: Gradniki in njihove lastnosti.



Slika 25: Hitrostno klikanje.

Koda obrazca (gumbom s sliko in obema gradnikoma *Panel* priredimo isto odzivno metodo *Gumb_Click*):

```
public partial class FIgra : Form
{
    public FIgra()
    {
        InitializeComponent();
    }
    int stevec;//števec klikov
    //spremenljivki tipa DateTime, ki onačujeta začetek in konec klikanja
    DateTime zacetek, konec;
    Random naklj = new Random();//generator naključnih števil
    private void FIgra_Load(object sender, EventArgs e)
    {
        OnemogociGumbe();//vsi štirje gumbi so na začetku onemogočeni
    }
    private void bZacetek_Click(object sender, EventArgs e)
    {
        (sender as Button).Visible = false; //skrijemo gumb
        OnemogociGumbe();
        MessageBox.Show("Pričetek igre. Igraš tako, da klikaš izbrane slike!
                         Igra se konča po 20 izbirah!", "Začetek");
        {
            zacetek = DateTime.Now;
            stevec = 0;//začetna vrednost števca klikov
        }
```

```
        OmogociNakljucniGumb();
    }
    private void OmogociNakljucniGumb()
    {
        int izbran = naklj.Next(1,5);//naključno število med 1 in 4
        //omogočimo gumb, ki ustreza naključnemu številu
        if (izbran == 1) button1.Enabled = true;
        else if (izbran == 2) button2.Enabled = true;
        else if (izbran == 3) button3.Enabled = true;
        else if (izbran == 4) button4.Enabled = true;
    }
    private void OnemogociGumbe()
    {
        button1.Enabled = false; //vse štiri gumbe deaktiviramo
        button2.Enabled = false;
        button3.Enabled = false;
        button4.Enabled = false;
    }
    /*metodo Gumb_Click priredimo vsem štirim gumbom s sliko, poleg tega pa
      še obem gradnikom Panel*/
    private void Gumb_Click(object sender, EventArgs e)
    {
        /*če je uporabnik zgrešil sliko (sliko predstavlja aktivni gumb, vse
          ostalo pa je izven slike) in se je igra že začela*/
        if (!(sender is Button)&& bZacetek.Visible==false)
        {
            MessageBox.Show("Zgrešil si gumb, število uspešnih zadetkov se
                             izniči! ", "Ponovni začetek!");
            stevec = 0;
        }
        else if (sender is Button) //če je bil kliknjen gumb
                {
                    /*preverimo, če je bil kliknjen pravi gumb (tisti, ki ima
                      lastost enabled nastavljeno na true)*/
                    if ((sender as Button).Enabled == true)
                    {
                        //izbrani gumb onemogočimo
                        (sender as Button).Enabled = false;
                        stevec++;//povečamo števec uspešnih klikov
                        if (stevec == 20)
                        {
                            konec= DateTime.Now;//čas,potreben za 20 zadetkov
                            //razlika začetnega in končnega časa
                            TimeSpan porabljenCas = konec - zacetek;
                            MessageBox.Show("Igra je končana!\n\nPotreboval
si " + porabljenCas.TotalSeconds + " sekund");
                        }
                        else
                        {
                            OmogociNakljucniGumb();//prikaz naključnega gumba
                        }
                    }
                }
```

```
    }
    private void bZapri_Click(object sender, EventArgs e)
    {
        Close(); //zapremo obrazec
    }
}
```

V zgornjem projektu smo prikazali enostaven primer uporabe operatorjev *is* in *as*. Koda je zato krajša in bolj razumljiva. Obenem smo ponovili še osnovni namen razredov *DateTime* in *TimeSpan*.

Kontrola uporabnikovih vnosov – validacija

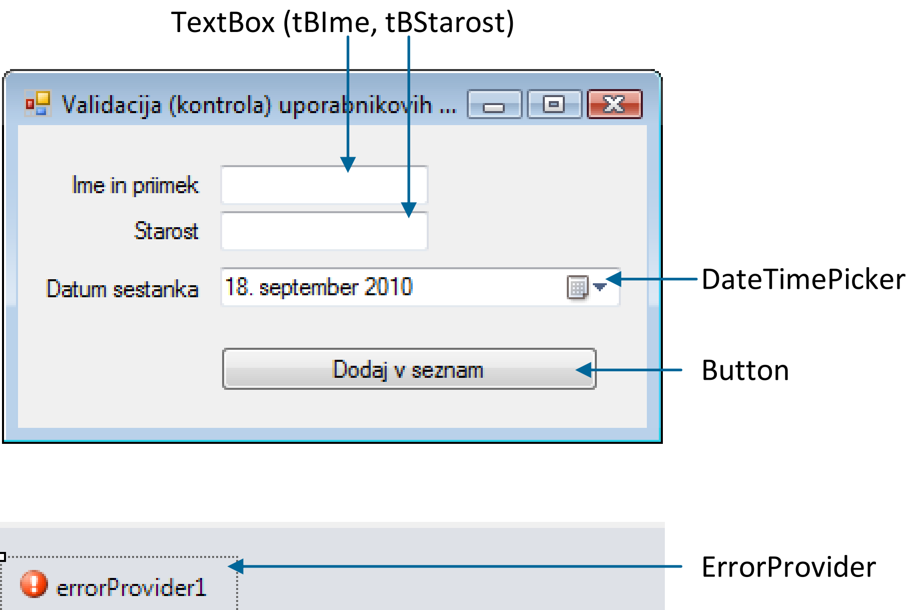
Pred uporabo ali pa pred shranjevanjem podatkov, ki jih uporabnik vnese v gradnike na obrazcu, je potrebno preveriti pravilnost (smiselnost) le-teh. Če podatki niso smiselni, je potrebno na to uporabnika opozoriti in mu pri velikem številu vnosov tudi vizuelno pokazati na gradnik, kjer je vnos nepravilen.

Gradniki, namenjeni uporabnikovim vnosom ali izbiram (npr. *TextBox*, *DateTimePicker*, *RadioButton*, *CheckBox*, *ComboBox*, …) imajo lastnost *CausesValidation*, ki je privzeto nastavljena na *True*. Ta lastnost pomeni, da vsak uporanikov premik iz tega gradnika na nekega drugega sproži dogodek *Validating*. Če to lastnost nastavimo na *False*, se dogodek ne bo izvedel in tako preverjanja ne bo. V ogrodje dogodka *Validating* zapišemo kodo, ali pa klic metode, ki preverja pravilnost vnosa. Dogodek *Validating* lahko zapišemo vsakemu gradniku posebej, na koncu pa še obrazcu, ali pa le obrazcu (odvisno od namena obrazca).

Obvestilo, kakšen vnos je potreben, pa lahko zapišemo tudi v polje *error on errorPrivider* (a le, če smo na obrazec postavili gradnik *ErrorProvider*): v tem primeru se bo že po prikazu obrazca ob gradniku prikazala ikona (privzeto klicaj na rdečem, okroglem ozadju). Če se z miško postavimo na to ikono, bomo v okvirčku dobili obvestilo, ki smo ga zapisali v polje e*rror on errorProvider*.

Napoved sestanka

Ustvarimo obrazec, ki bo od uporabnika zahteval vnos imena in priimek, njegovo starost in datum nekega sestanka. Vsi trije podatki so obvezni, poleg tega mora biti uporabnik polnoleten, izbrani datum pa ne sme biti na soboto ali nedeljo. Kontrolo vnosov bomo realizirali s pomočjo metod gradnika *ErrorProvider*, ki ga vežemo na obrazec. Besedilo o napaki zapišemo v metodo *SetError* tega gradnika, ki ima dva parametra: prvi parameter je ime gradnika, za katerega pišemo odzivno metodo, drugi pa sporočilo o napaki!.



Slika 26: Kontrola uporabnikovih vnosov.

Gradniki in njihove lastnosti

Gradnik	Lastnost	Nastavitev	Opis
FValidacija	FormBorderStyle	FixedToolWindow	Uporabnik velikosti okna ne more spreminjati
tBStarost	TextAlign	Right	Vnesena starost bo desno poravnana.
errorProvider1	ContainerControl	FValidacija	Nadrejeni gradnik (običajno, pa tudi v našem primeru, je to obrazec), na katerem bo prikazana ikona z opozorilom, da je vnos napačen.

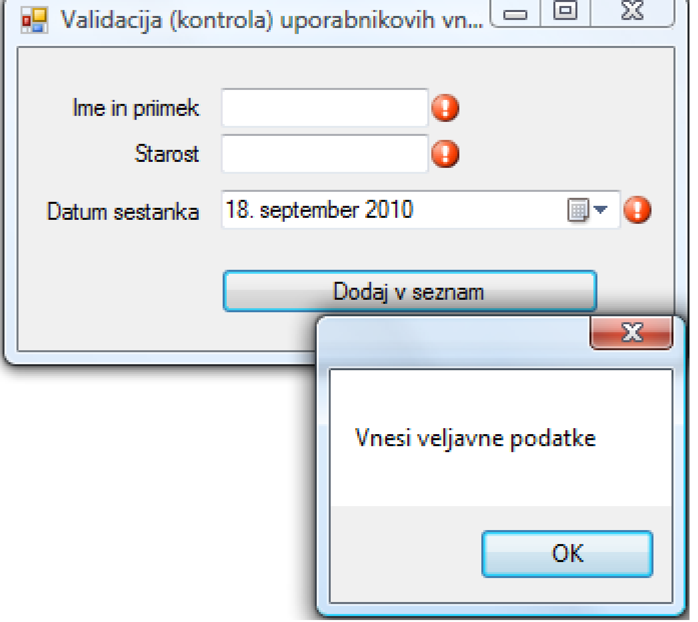
Tabela 7: Gradniki na obrazcu *FValidacija*.

```
public FValidacija()
{
    InitializeComponent();
    tbStarost.Text = "";
}
//logične spremenljivke za preeverjanje pravilnih vnosov podatkov
bool kontrolaImena = false, kontrolaStarosti = false, kontrolaDatuma = false;
//odzivna metoda Validating gradnika textBox1
private void tBIme_Validating(object sender, CancelEventArgs e)
{
```

```
    if (tBIme.Text == "")
    {
        /*Če uporabnik ne bo vnesel imena, se bo ob imenu pokazala oznaka s
klicajem. Če se bomo z miško postavili na to oznako, se pod njim v okvirčku
pojavi besedilo, ki ga zapišemo kot drug parameter metode SetError */
        errorProvider1.SetError(tBIme, "Vnesi svoje ime");
        kontrolaImena = false;
    }
    else
    {
        errorProvider1.SetError(tBIme, "");
        kontrolaImena = true;
    }
}
//odzivna metoda Validating gradnika textBox2
private void tBStarost_Validating(object sender, CancelEventArgs e)
{
    kontrolaStarosti = false;
    if (tbStarost.Text == "")
    {
        errorProvider1.SetError(tbStarost, "Vnesi starost");
    }
    else
    {
        errorProvider1.SetError(tbStarost, "");
        try
        {
            int temp = int.Parse(tbStarost.Text);
            errorProvider1.SetError(tbStarost, "");
            if (temp < 18)
            {
                errorProvider1.SetError(tbStarost, "Za reševanje testa moraš
biti star najmanj 18 let");
            }
            else
            {
                errorProvider1.SetError(tbStarost, "");
                kontrolaStarosti = true;
            }
        }
        catch
        {
            errorProvider1.SetError(tbStarost, "Starost mora biti številka");
        }
    }
}
//odzivna metoda Validating gradnika
private void dateTimePicker1_Validating(object sender, CancelEventArgs e)
{
    kontrolaDatuma =false;
    //Če je izbrani datum sobota ali nedelja, opozorimo uporabnika
    if ((dateTimePicker1.Value.DayOfWeek == DayOfWeek.Sunday) ||
(dateTimePicker1.Value.DayOfWeek == DayOfWeek.Saturday))
```

```
    {
        errorProvider1.SetError(dateTimePicker1, "Sestanek ne more biti 
organiziran med vikendom. Izberi prosim dan med tednom!");
    }
    else
    {
        errorProvider1.SetError(dateTimePicker1, "");
        kontrolaDatuma = true;
    }
}
//Sestanek bo organiziran le v primeru, da so vsi podatki na obrazcu venseni 
pravilno
private void ValidateForm()
{
    if (kontrolaImena && kontrolaStarosti && kontrolaDatuma)
        MessageBox.Show("Srečanje bo organizirano!");
    else
        MessageBox.Show("Vnesi veljavne podatke");
}

private void Shrani_Click(object sender, EventArgs e)
{
    ValidateForm();
}

private void FSestanek_Activated(object sender, EventArgs e)
{
    tbStarost.Text = tBIme.Text;
}
```

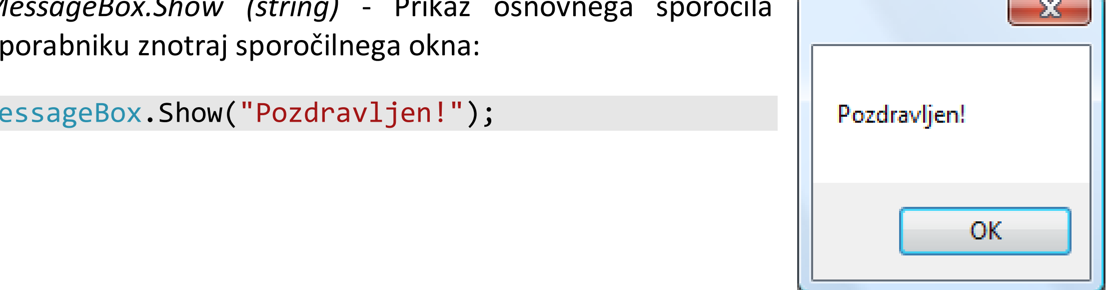
V projektu smo prikazali le "masko" za vnos podatkov o nekem sestanku, dejansko dodajanje v seznam sestankov pa bi morali še napisati, npr. v nekem nadrejenem obrazcu, to je obrazcu iz katerega smo odprli obrazec *FValidacija*.



Slika 27: Če so vnosi napačni (oz. vnosa ni), se ob gradniku pojavi ikona z opozorilom.

Sporočilno okno MessageBox

Sporočilno okno *MessageBox* je namenjeno posredovanju sporočil. Obenem je to tudi pogovorno okno namenjeno uporabnikovim odločitvam. Okno lahko prikaže sporočila v številnih variantah. V dosedanjih primerih smo ga uporabljali le v njegovi najenostavnejši obliki, z enim samim parametrom. Prikažemo ga s pomočjo metode *Show*. Obstaja kar 21 različnih načinov (preobtežitev) uporabe te metode. Ogledali pa si bomo le najpomembnejše.



- *MessageBox.Show (string)* - Prikaz osnovnega sporočila uporabniku znotraj sporočilnega okna:

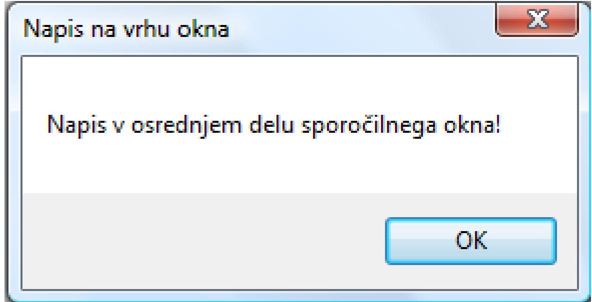
```
MessageBox.Show("Pozdravljen!");
```


Slika 28: Osnovna uporaba sporočilnega okna *MessageBox*.

V oknu se pokaže obvestilo, ki smo zapisali v oklepaju metode *Show*, pod besedilom pa je gumb z napisom *OK*. Ob kliku na ta gumb se sporočilno okno zapre.

- *MessageBox.Show (string, string)* - Prikaz poljubnega teksta v sporočilnem oknu in še poljubnega teksta v naslovni vrstici okna.

```
MessageBox.Show("Napis v osrednjem delu sporočilnega okna!", "Napis na
vrhu okna");
```



Slika 29: Sporočilno okno z dvema parametroma tipa string.

- *MessageBox.Show (string, string, MessageBoxButtons)* - Sporočilno okno s tekstom v oknu, napisom na oknu in odločitvenimi gumbi.

Prva dva parametra sta dva niza (tip *string*). Prvi parameter predstavlja napis v oknu, drugi pa napis na oknu. Tretji parameter je tipa *MessageBoxButtons* in določa število in vrsto gumbov, ki bodo prikazani v oknu. To je v bistvu naštevni tip s konstantami, ki določajo kateri gumbi se bodo prikazali v sporočilnem oknu. Napisi na gumbih so odvisni od jezikovne variante operacijskega sistema. Vse možne konstante in njihova razlaga, so zbrane v naslednji tabeli:

MessageBoxButtons	Razlaga (SLO verzija operacijskega sistema in ANJ verzija operacijskega sistema)
AbortRetryIgnore	Sporočilno okno vsebuje gumbe *Prekini*, *Poskusi znova*, in *Prezri* (*Abort*, *Retry*, *Ignore*).
OK	Sporočilno okno vsebuje gumb *V redu* (OK).
OKCancel	Sporočilno okno vsebuje gumba *V redu* in *Prekliči* (*OK*, *Cancel*).
RetryCancel	Sporočilno okno vsebuje gumba *Poskusi znova*, in *Prekliči* (Retry, *Cancel*).
YesNo	Sporočilno okno vsebuje gumba *Da* in *Ne* (*Yes*, *No*).
YesNoCancel	Sporočilno okno vsebuje gumbe *Da*, *Ne* in *Prekliči* (*Yes*, *No*, *Cancel*)

Tabela 8: Vrednosti naštevnega tipa *MessageBoxButtons.*

MessageBox je razred, metoda *Show* pa statična metoda tega razreda, ki je naštevnega tipa (*enum*) *DialogResult*. Vrednost, ki jo metoda *Show* vrne je torej tipa *DialogResult*, odvisna pa je od uporabnikovega klika na določen gumb. Od tega pa je seveda odvisno nadaljevanje aplikacije (za kar moramo poskrbeti z ustrezno kodo). Vrednosti tipa *DialogResult* so zbrane v naslednji tabeli:

DialogResult	Razlaga
Abort	Prekini (Abort).
Cancel	Prekliči (Cancel).
Ignore	Prezri (Ignore).
No	Ne (No).
OK	V redu (OK).
Retry	Poskusi znova (Retry).
Yes	Da (Yes).

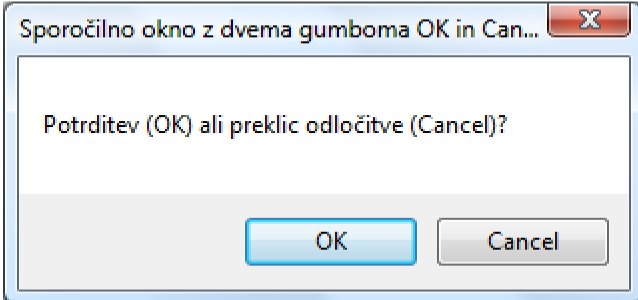
Tabela 9: Vrednosti tipa *DialogResult*.

Kadar sporočilno okno vsebuje več kot en gumb, moramo seveda predvideti klik na kateregakoli od teh gumbov in ustrezno reagirati v aplikaciji. Tule je primer uporabe parametra *MessageBoxButtons* z gumboma *OK* in *Cancel*:

```
/*ker metoda Show vrne vrednost tipa DialogResult jo lahko uporabimo
takole*/

DialogResult odlocitev = MessageBox.Show("Potrditev (OK) ali preklic
odločitve (Cancel)?", "Sporočilno okno z dvema gumboma OK in Cancel!",
MessageBoxButtons.OKCancel);

if (odlocitev == DialogResult.OK)
   //stavki, ki se izvedejo, če je uporabnik kliknil gumb OK
else
   //stavki, ki se izvedejo, če je uporabnik kliknil gumb Cancel
```

ali krajše (in lepše) takole:

```
if (MessageBox.Show("Potrditev (OK) ali preklic odločitve (Cancel)?",
"Sporočilno okno z dvema gumboma OK in Cancel!",
MessageBoxButtons.OKCancel) == DialogResult.OK)
    //stavki, ki se izvedejo, če je uporabnik kliknil gumb OK
else
    //stavki, ki se izvedejo, če je uporabnik kliknil gumb Cancel
```



Slika 30: Uporaba parametra *MessageBoxButtons* v sporočilnem oknu.

- *MessageBox.Show (string, string, MessageBoxButtons, MessageBoxIcon)* - Sporočilno okno s tekstom v oknu, napisom na oknu, odločitvenimi gumbi in *ikono* v oknu.

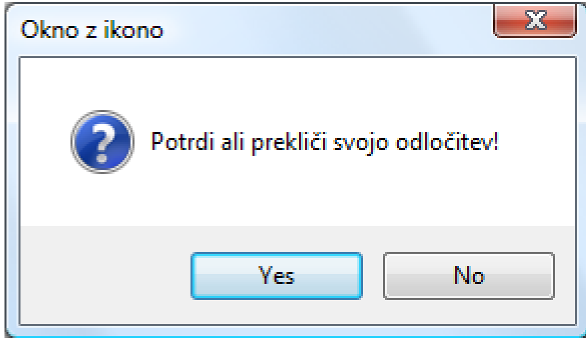
V sporočilnem oknu lahko prikažemo tudi eno od vnaprej pripravljenih ikon (vprašaj, klicaj, napaka, opozorilo). Prvi trije parametri metode so enaki kot v prejšnjem primeru, četrti parameter pa je tipa *MessageBoxIcon*. To je naštevni tip s konstantami, ki določajo katera ikona se bo prikazala v sporočilnem oknu. Vse možne konstante, skupaj z razlago so zbrane v naslednji tabeli:

Oznaka ikone	Razlaga
Asterisk	Sporočilno okno vsebuje grafični simbol z malo črko i v sredini.
Error	Sporočilno okno vsebuje grafični simbol z znakom X na rdeči podlagi.
Exclamation	Sporočilno okno vsebuje grafični simbol z znakom ! (klicaj) na rumeni

	podlagi.
Hand	Sporočilno okno vsebuje grafični simbol z znakom X na rdeči podlagi.
Information	Sporočilno okno vsebuje grafični simbol z malo črko i.
None	Sporočilno okno ne vsebuje grafičnega simbola.
Question	Sporočilno okno vsebuje grafični simbol z znakom ? (vprašaj) na modri podlagi.
Stop	Sporočilno okno vsebuje grafični simbol z znakom X na rdeči podlagi.
Warning	Sporočilno okno vsebuje grafični simbol z znakom ! (klicaj) na rumeni podlagi.

Tabela 10: Vrednosti naštevnega tipa *MessageBoxIcon*.

```
//klic sporočilnega okna z gumboma Yes, NO in ikono z vprašajem
if (MessageBox.Show("Potrdi ali prekliči svojo odločitev!", "Okno z
ikono", MessageBoxButtons.YesNo, MessageBoxIcon.Question) ==
DialogResult.Yes)
    //stavki, ki se izvedejo, če uporabnik klikne gumb Yes
else
    //stavki, ki se izvedejo, če uporabnik klikne gumb No
```



Slika 31: Sporočilno okno z ikono.

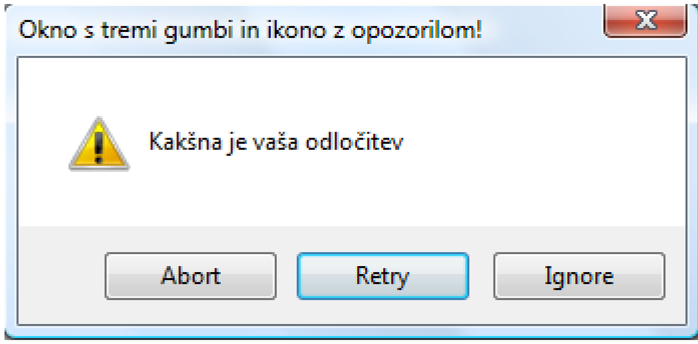
- *MessageBox.Show(string, string, MessageBoxButtons, MessageBoxIcon, MessageBoxDefaultButton)*

Kadar je v sporočilnem oknu več kot en gumb, lahko vnaprej določimo privzeti gumb, to je gumb, ki je vnaprej označen (aktiven). Peti parameter (aktivni gumb v oknu) je v v tem primeru parameter tipa *MessageBoxDefaultButton*, ki predstavlja oznako enega izmed treh možnih gumbov v sporočilnem oknu. Gumb, naveden kot peti parameter, bo izbrani gumb (bo torej aktiven in ima fokus), ko se pogovorno okno prikaže.

Če torej želimo na potek nadaljevanja programa vplivati preko takega sporočilnega okna, bi zapisali

```
//Sporočilno okno s tremi gumbi, izbrani (aktivni) gumb je gumb Retry
```

```
switch (MessageBox.Show("Kakšna je vaša odločitev", "Okno s tremi gumbi
in ikono z opozorilom!", MessageBoxButtons.AbortRetryIgnore,
MessageBoxIcon.Warning, MessageBoxDefaultButton.Button2))
{
    case DialogResult.Abort: /*stavki, ki se izvedejo, če uporabnik
                               klikne gumb Abort*/
        break;
    case DialogResult.Retry: /*stavki, ki se izvedejo, če uporabnik
                               klikne gumb Retry*/
        break;
    case DialogResult.Ignore: /*stavki, ki se izvedejo, če uporabnik
                                klikne gumb Ignore*/
        break;
}
```



Slika 32: Sporočilno okno s tremi gumbi, ikono in izbranim aktivnim gumbom.

- *MessageBox.Show (string, string, MessageBoxButtons, MessageBoxIcon, MessageBoxDefaultButton, MessageBoxOptions)*

Sporočilno okno s tekstom v oknu, napisom na oknu, odločitvenimi gumbi, vrsto ikone, privzetim gumbom in *opcijami*. Šesti parameter (možnosti v oknu) je parameter tipa *MessageBoxOptions*. To je oznaka ene izmed štirih posebnih možnosti, s katerimi povemo, kako naj se prikaže sporočilo v sporočilnem oknu. Seznam vseh možnosti, ki jih lahko zapišemo kot šesti parameter je v naslednji tabeli:

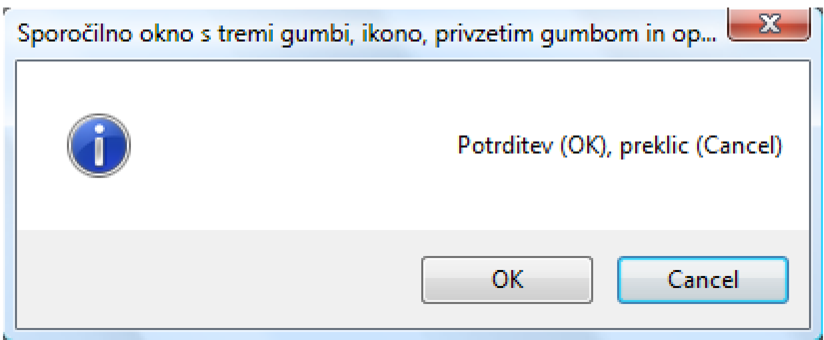
Oznaka Parametra	Razlaga
DefaultDesktopOnly	Sporočilno okno je prikazano na aktivnem namizju.
RightAlign	Tekst sporočila je desno poravnan.
RtlReading	Nastavitev določa, da je vsebina sporočilnega okna prikazana od desne proti levi (besedilo je na levi, ikona pa na desni.
ServiceNotification	Sporočilno okno je prikazano na aktivnem namizju.

Tabela 11: Tabela možnih nastavitev v oknu *MessageBox*

S kodo

```
if (MessageBox.Show("Potrditev (OK), preklic (Cancel)", "Sporočilno
okno s tremi gumbi, ikono, privzetim gumbom in opcijo",
MessageBoxButtons.OKCancel,MessageBoxIcon.Information,MessageBoxDefault
Button.Button2,MessageBoxOptions.RightAlign)==DialogResult.Cancel)
{  //stavki, ki se izvedejo, če je uporabnik kliknil gumb OK}
else
{  //stavki, ki se izvedejo, če je uporabnik kliknil gumb Cancel}
```

torej dosežemo, da se okno pojavi v taki obliki:



Slika 33: Sporočilno okno z desno poravnavo teksta.

- *MessageBox.Show (string, string, MessageBoxButtons, MessageBoxIcon, MessageBoxDefaultButton, MessageBoxOptions,string FileHelpPath)*

Sporočilno okno s tekstom v oknu, napisom na oknu, odločitvenimi gumbi, vrsto ikone, privzetim gumbom, opcijami in dodatnim gumbom za pomoč oz. ime datoteke s pomočjo.

Če namesto sedmega parametra zapišemo le besedico *true*, bo v sporočilnem oknu prikazan gumb z napisom *Help* (oz. Pomoč). Ob kliku na ta gumb se bo izvedel dogodek *HelpRequested*, ki pa ga moramo seveda prej pripraviti. Za vajo v ta namen pripravimo dogodek obrazca *Form1_HelpRequested*, v katerega zapišimo le klic enostavnega sporočilnega okna.

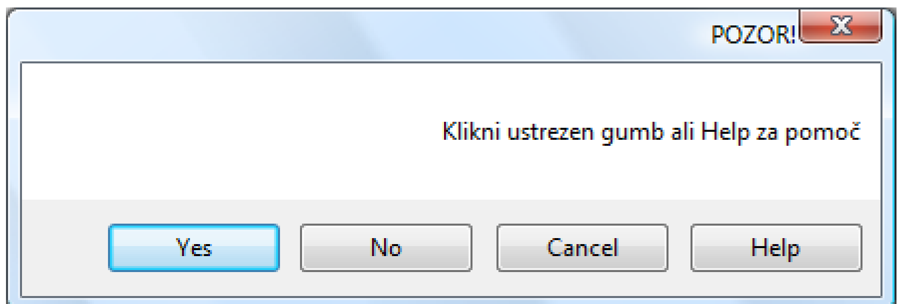
```
private void Form1_HelpRequested(object sender, HelpEventArgs hlpevent)
{
    MessageBox.Show("Pomoč...");
}
```

Če se v metodi *Show* sporočilnega *MessageBox* ne želimo navesti katerega od parametrov (razen prvih dveh, ki sta tipa niz), namesto njih zapišemo vrednost 0. Takrat tega parametra ne upoštevamo oz. upoštevamo privzete vrednosti

Klic sporočilnega okna lahko sedaj zapišemo takole:

```
/*v metodi testiramo, kateri gumb za zapiranje sporočilnega okna je
  kliknil uporabnik*/
private void button1_Click(object sender, EventArgs e)
{
```

```
    switch (MessageBox.Show("Klikni ustrezen gumb ali Help za pomoč",
"POZOR!", MessageBoxButtons.YesNoCancel, 0, 0,
MessageBoxOptions.RightAlign, true))
    {
        case DialogResult.Yes: /*stavki, ki se izvedejo, če uporabnik
                                 klikne gumb Yes*/
            break;
        case DialogResult.No: /*klik na gumb No*/
            break;
        case DialogResult.Cancel: /*klik na gumb Prekliči*/
            break;
    }
}
```



Slika 34: Prikaz gumba *Help* (Pomoč).

Kot sedmi parameter običajno napišemo ime neke datoteke s pomočjo. V sporočilnem oknu bo v tem primeru prikazan dodaten gumb z napisom *Help* (oz. *Pomoč*, če imamo slovenski operacijski sistem*)*. Ob kliku na ta gumb se prikaže okno z vsebino datoteke tipa *.chm*, katere ime smo zapisali kot sedmi parameter.

```
//POZOR: datoteka Help.chm mora obstajati
switch (MessageBox.Show("Klikni ustrezen gumb ali Help za pomoč",
"POZOR!", MessageBoxButtons.YesNoCancel,0,0,0,
@"c:\Ikone\Ikone\standard-software-icons\Help.chm"))
{
    case DialogResult.Yes: /*stavki, ki se izvedejo, če uporabnik
                             klikne gumb Yes*/
                           break;
    case DialogResult.No: /*klik na gumb No*/
                          break;
    case DialogResult.Cancel: /*klik na gumb Cancel*/
                              break;
}
```

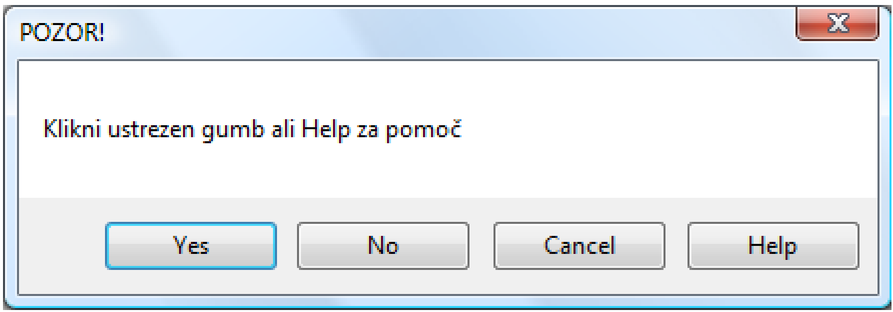
Pa še eno pomembno opozorilo, čeprav o programskem dodajanju in odvzemanju odzivov na dogodke v tej literaturi še ni bilo govora: v kolikor smo na obrazcu, v okviru katerega izvajamo to sporočilno okno, že ustvarili odzivni dogodek *HelpRequested* (npr *Form1_HelpRequested*), moramo pred prikazom sporočilnega okna odziv na ta dogodek začasno odstraniti s stavkom

```
this.HelpRequested -= new
System.Windows.Forms.HelpEventHandler(this.Form1_HelpRequested);
```

Če tega ne storimo, sta uporabnikovem kliku na gumb *Help* prirejena dva enaka dogodka. Posledica tega je hkratno odpiranje sporočilnega okna in še okna z vsebino datoteke s pomočjo.

Po zaprtju sporočilnega okna ne smemo pozabiti dogodka *HelpRequested* vključiti nazaj s stavkom

```
this.HelpRequested += new
System.Windows.Forms.HelpEventHandler(this.Form1_HelpRequested);
```



Slika 35: Prikaz gumba in datoteke s pomočjo.

- *MessageBox.Show (string , string, MessageBoxButtons , MessageBoxIcon , MessageBoxDefaultButton, MessageBoxOptions, string, HelpNavigator)*

  Sporočilno okno s tekstom v oknu, napisom na oknu, odločitvenimi gumbi, vrsto ikone, privzetim gumbom, opcijami, nizom s katerim podamo ime datoteke s pomočjo in parametrom s katerim povemo kateri element datoteke s pomočjo bo prikazan.

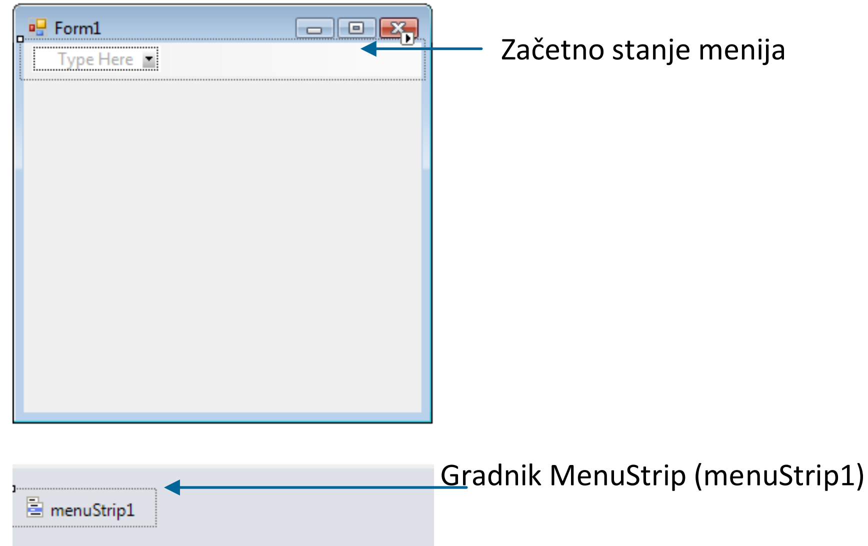
Opisanih in na primerih razloženih je bilo le nekaj najpogosteje uporabljenih oblik metode *Show* razreda *MessageBox*.

Delo z enostavnimi in sestavljenimi meniji, orodjarna

Meniji so standardni elementi okenskih programov. V menijih zapisane vrstice predstavljajo ukaze in gume, ki jih lahko aktiviramo z miško ali pa s tipkovnico. Poznamo dve vrsti menijev: glavnega, ki ga navadno postavimo pod gornji rob obrazca in lebdečega, ki ga odpre klik desnega gumba miške na določenem predmetu. Gradnika, namenjena oblikovanju menijev, sta *MenuStrip* – glavni meni in *ContextMenuStrip* – lebdeči meni.

Glavni meni – MenuStrip (nadgradnja)

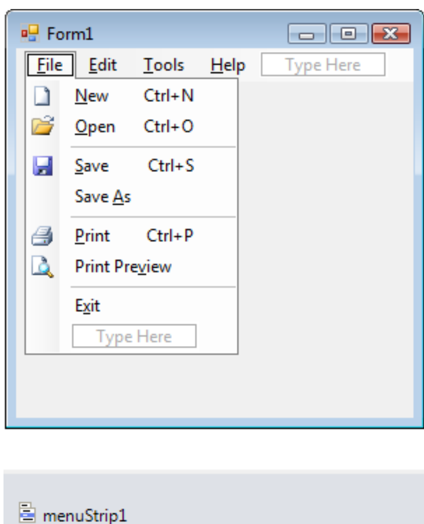
Gradnik *MenuStrip*, ki ga potrebujemo za ustvarjanje glavnega menija, se nahaja v oknu *Toolbox* v skupini gradnikov *Menus & Toolbars*. To je nevizuelni gradnik: ko ga postavimo na obrazec se namesti v polje pod obrazcem, na vrhu obrazca pa se pokaže začetno stanje menija.



Slika 36: Ustvarjanje glavnega menija – na obrazec postavimo gradnik *MenuStrip*.

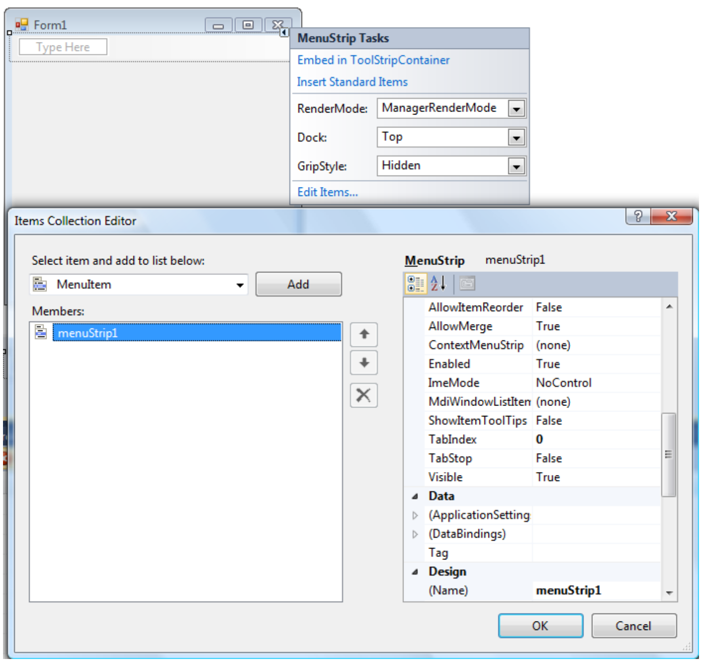
Z oblikovanjem menija pričnemo na več načinov:

- kliknemo v okno menija z napisom *Type Here* in zapišemo tekst, ki ga želimo imeti v meniju. Tekst menija nato po želji dodajamo v horizontalni ali pa v vertikalni smeri. Urejevalnik menija je videti kot pravi meni, le da ima nakazana prazna mesta, kamor je mogoče pisati nove postavke. Več o menijih in o sestavljenih menijih bo prikazano v posebnem poglavju;
- kliknemo na puščico v zgornjem desnem robu začetnega menija: odpre se pogovorno okno *MenuStrip Tasks*, kjer pa imamo zopet na voljo več možnosti. Najpomembnejše so
    - *Embed in ToolStrip Container*: ob izbiri se bo v hipu zgradil celotni meni z najpogostejšimi opcijami, ki se v menijih uporabljajo, a meni bo zgrajen znotraj gradnika *ToolStripContainer*. Takega načina ustvarjanja menija zaenkrat ne bomo uporabljali;
    - *InsertStandard Items*: v tem primeru se bo v hipu zgradil celotni meni z najpogostejšimi opcijami, ki se v menijih uporabljajo, obenem pa bodo v meniju tudi sličice, separatorji …) ;



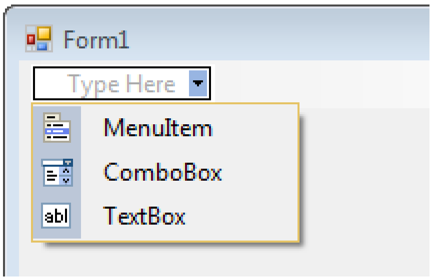
Slika 37: Glavni meni s standardnimi opcijami.

- *Edit Items*: posamezne postavke menija, tako v vertikalni, kot v horizontalni smeri, bomo oblikovali sami s pomočjo *okna Items Collection Editor*;



Slika 38: Ročno oblikovanje menija s pomočjo urejevalnika menija.

Preden začnemo pisati besedilo nove postavke menija, lahko na desni strani postavke odpremo spustni seznam in izberemo kakšen tip postavke želimo (*MenuItem* - običajna vrstica menija, *ComboBox* - spustni seznam, ali pa *TextBox* - okno z besedilom).

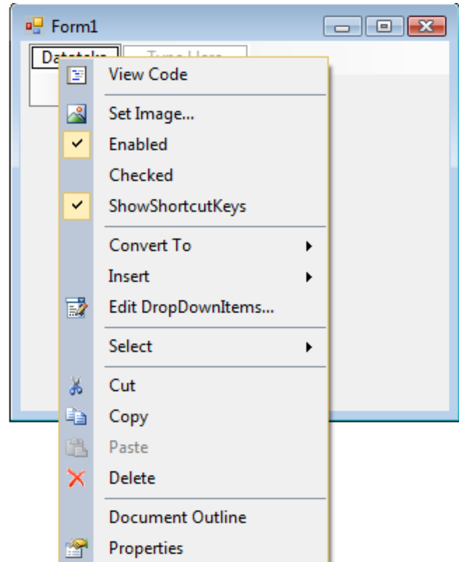


Slika 39: Pri ustvarjanju menija lahko izbiramo med tremi možnostmi.

Vsaka postavka v meniju ima svojo kartico lastnosti, ki so prikazane v oknu *Properties*. Posamezno postavko menija najpogosteje predstavlja *MenuItem*. Najbolj značilna lastnost menijske postavke je njen napis (lastnost *Text*), iz katere *Visual C#* tudi izpelje ime postavke (lastnost *Name*). Če smo npr. za neko postavko napisali ime *Konec*, je njeno programsko ime *KonecToolStripMenuItem*.

Če pred ime postavke menija, ali pa pred katerikoli znak v menijski postavki, zapišemo znak &, bo ta znak v meniju podčrtan. To pomeni, da lahko do te postavke dostopamo tudi s kombinacijo tipk *Alt+Znak*.

Z desnim klikom nad posamezno postavko menija na obrazcu se prikaže *PopUp* meni s številnimi opcijami.



Slika 40: *Pop Up* meni za oblikovanje menija.

Pomen posameznih postavk:

- *View Code*: preklop v urejevalniško okno s kodo;
- *Set Image:* izbira slike (ikone), ki naj se prikaže ob postavki menija;
- *Enabled*: postavka menija omogočena ali onemogočena: če je lastnost onemogočena, se dogodek, ki je tej postavki prirejen, ne bo izvedel;

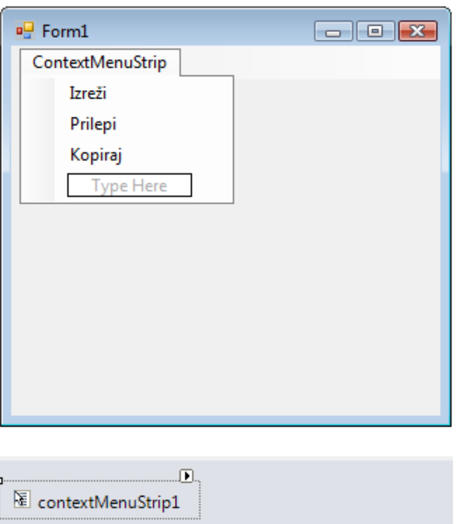
- *Cheched*: če jo označimo se na levi strani prikaže kljukica: postavka menija se lahko sedaj odziva različno glede na to, ali je postavka odkljukana ali ne;
- *ShowShortcutKeys*: vklop/izklop prikaza kombinacije tipk za dostop do te postavke menija preko tipkovnice (v tem primeru moramo v oknu *Properties* gradniku določiti lastnost *ShortcutKeys* in izbrati ustrezno kombinacijo tipk);
- *Convert To*: pretvorba postavke menija v neko drugo obliko (vsaka postavka v meniju je lahko prikazana na štiri načine: kot *MenuItem*, *ComboBox*, *Separator* ali kot *TextBox*);
- *Insert*: vrivanje nove postavke menija;
- *Edit DropDownItems…*: urejanje postavk v vertikalni smeri (podmenija lahko ustvarimo tudi s kombinacijo tipk *Ctrl + →*);
- *Select*: izbira te opcije nam ponudi seznam gradnikov, v katerem izberemo, kateri od njih naj bo izbran;
- *Delete*: brisanje že narejene postavke;
- *Cut*: izreži izbrano opcijo menija;
- *Copy*: kopiranje izbrane opcije menija;
- *Delete*: brisanje izbrane opcije menija;
- *Document Outline*: prikaz drevesne strukture celotnega menija v novem oknu, ki se običajno prikaže pod oknom *Toolbox*;
- *Properties*: prikaz lastnosti izbrane opcije menija.

Ločilno črto (separator) v meniju lahko naredimo tudi tako, da namesto vnosa imena neke postavke zapišemo le znak pomišljaj "-".

Vsaki postavki v meniju priredimo odzivno metodo tako, da jo najprej izberemo, nato v oknu *Properties* kliknemo hitri gumb za prikaz dogodkov (*Events*) in končno izberemo ustrezni dogodek. Običajno bo to dogodek *Click* ali pa *DoubleClick*. Dvokliknemo v prazno polje ustreznega dogodka, da nam C# pripravi ogrodje metode, nato pa končno zapišemo ustrezne stavke odzivne metode. Ime metode bo v tem primeru privzeto (npr. *toolStripMenuItem1_Click*). Dogodku pa seveda lahko priredimo tudi poljubno ime (bodisi tako, da pred dvoklikom v oknu *Propertires→Events* zapišemo ime dogodka, ali pa v urejevalniškem oknu čez staro ime zapišemo novo ime in spremembo potrdimo v celotnem projektu).

Lebdeči (Pop Up) meni - ContextMenuStrip

Lebdeči meni se v delujoči aplikaciji pokaže, ko na nek gradnik kliknemo z desnim gumbom miške. Priredimo ga lahko kateremukoli gradniku, ki smo ga že postavili na obrazec. To storimo tako, da ime lebdečega menija izberemo kot lastnost *ContextMenuStrip* izbranega gradnika (npr. gumba, vnosnega polja …). Seveda pa moramo prej lebdeči meni pripraviti. Izdelava je podobna izdelavi glavnega menija. Razlika je le tem, da uporabimo gradnik *ContextMenuStrip*, ki se tako kot gradnik *MenuStrip* nahaja v oknu *Toolbox* v skupini gradnikov *Menus & Toolbars*. Tudi ta gradnik je nevizuelni gradnik. Ko ga postavimo na obrazec, se namesti v polje pod obrazcem, na vrhu obrazca pa se pokaže začetno stanje menija.

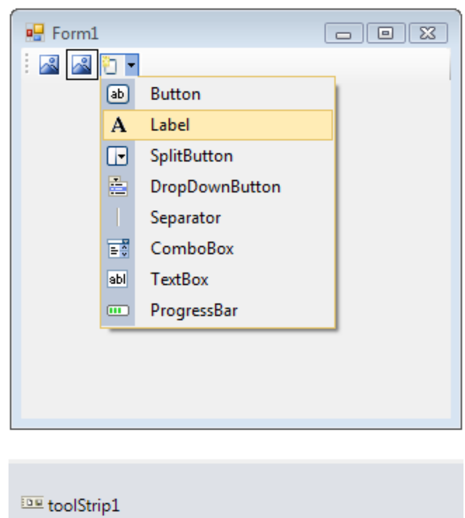


Slika 41: Ustvarjanje lebdečega menija.

Lebdeči meni urejamo tako kot glavni meni: v prazno okno vpišemo besedilo postavke, nato pa se premaknemo navzdol, če želimo imeti naslednjo opcijo v navpični smeri, ali pa desno, če želimo v tej vrstici začeti nov podmeni. Vsaki postavki v lebdečem meniju priredimo odzivni dogodek tako, da jo najprej izberemo, nato v oknu *Properties* kliknemo hitri gumb za prikaz dogodkov (*Events*) in končno izberemo ustrezni dogodek. Običajno bo to dogodek *Click* ali pa *DoubleClick*. Dvokliknemo v prazno polje ustreznega dogodka, da nam *C#* pripravi ogrodje metode, nato pa končno zapišemo ustrezne stavke odzivne metode.

Orodjarna - ToolStrip

Orodjarna ali orodna vrstica (letvica) je vsebnik – gradnik, ki vsebuje druge gradnike. V oknu *Toolbox* je to gradnik *ToolStrip*, nahaja pa se v skupini gradnikov *Menus & Toolbars*. Na letvico lahko postavljamo gradnike tipa *Button*, *Label*, *SplitButton*, *DropDownButton*, *Separator*, *ComboBox*, *TextBox* in *ProgresBar*. Nove gradnike v orodjarno dodajamo tako, da odpremo spustni seznam na desni strani letvice in izberemo vrsto gradnika.

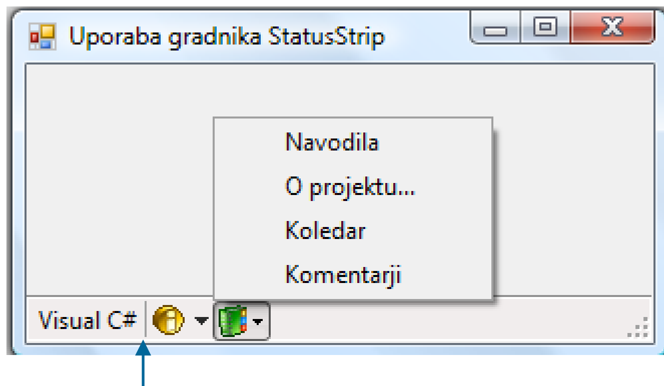


Novo postavko tipa *Button* v orodjarni lahko najhitreje dobimo tako, da nemsto odpiranja spustnega seznama nanj le kliknemo. Tako nastalim gumbom lahko poljubno spreminjamo lastnosti (npr. sličico na gumbu, …) in prirejamo odzivne metode (najpogosteje dogodek *Click*). Posamezne gradnike znotraj orodjarne lahko kadarkoli odstranimo (pobrišemo) ali pa jih preuredimo (primemo z miško in prenesemo na drugo pozicijo v orodjarni).

Slika 42: Oblikovanje orodjarne.

Gradnik StatusStrip

StatusStrip je gradnik, ki se običajno prikazuje na dnu obrazca. Namenjen je oblikovanju statusne vrstice na dnu okna. V gradniku ustvarjamo predalčke: to storimo tako, da odpremo spustni seznam znotraj tega gradnika in izberemo eno od ponujenih opcij (*StatusLabel* – napis, *ProgressBar*, *DropDownButton* in *SplitButton*). Predalčkom tipa *StatusLabel* običajno še priredimo lastnost *BorderSides – Right*, da dobimo robove. Delo s posameznimi predalčki je zelo podobno oblikovanju glavnega menija. Širina predalčkov (*size*) je lahko poljubna, le lastnost *AutoSize* posameznega predalčka moramo prej nastaviti na *False*. Kadar je predalček tipa *StatusLabel*, mu lahko določimo še dve posebni lastnosti: lastnost *isLInk* (če jo nastavimo na *True*, se obnaša kot bližnjica, npr. do spletne strani), in lastnost *Spring* (če jo nastavimo na *True*, se predalček s oznako razširi tako, da z ostalimi predalčki zavzema celotno širino obrazca).



Slika 43: Gradnik *StatusStrip* s tremi predalčki.

Do posameznega predalčka gradnika *StatusStrip* lahko dostopamo tudi programsko. Vsi skupaj sestavljajo zbirko predalčkov (*Items*), do posameznega pa dostopamo preko indeksa, npr. takole:

```
//prikaz teksta v prvem predalčku (indeks je 0) statusne vrstice
statusStrip1.Items[0].Text = "TŠC Kranj!";
```

Dialogi – okenska pogovorna okna

Poleg sporočilnega okna *MessageBox* vsebuje razvojno okolje *Visual C#* še kar nekaj koristnih in zelo uporabnih pogovornih oken. Standardna pogovorna okna najdemo v oknu *Toolbox* v skupini *Dialogs*. Če kateregakoli od teh gradnikov (dialogov) postavimo na obrazec se njegova "slika" pokaže v polju pod obrazcem. Tam ga potem lahko kadarkoli izberemo, ter mu prirejamo lastnosti in dogodke. S tem, ko pa smo ga postavili na obrazec, smo v projekt vključili objekt

ustreznega pogovornega okna. Imena tako ustvarjenih objektov so privzeta: (*colorDialog1*, *fontDialog1*, *openDialog1*…), do njihovih lastnosti in dogodkov pa tako kot pri vseh objektih dostopamo s pomočjo operatorja pika.

Osnovna pogovorna okna, ki jih pozna *Visual C#*, so naslednjih tipov:

- *ColorDialog*,
- FolderBrowserDialog,
- *FontDialog*,
- *OpenDialog* in
- *SaveDialog*.

Vsa pogovorna okna naštetih tipov odpiramo (prikažemo) z metodo *ShowDialog*. Okna so *modalna*, kar pomeni, da vračajo vrednost naštevnega tipa *DialogResult* (tako kot smo to že videli pri sporočilnem oknu *MessageBox*).

ColorDialog – pogovorno okno za izbiro barve

Pogovorno okno *ColorDialog* je namenjeno izbiri barve. Če uporabnik v tem oknu izbere poljubno barvo in okno zapre z gumbom *OK* (*V redu*), se izbrana barva shrani v lastnost okna *Color*. Na ta način lahko uporabniku omogočimo, da po svojih željah nastavi barvo, ki jo bo v programu uporabil na različne načine (lahko bo to vrednost nekega polja v bazi, ozbrana barva lahko povzroči spremembo barve ozadja, od izbire barve je lahko odvisna tudi razvejitev programa…).

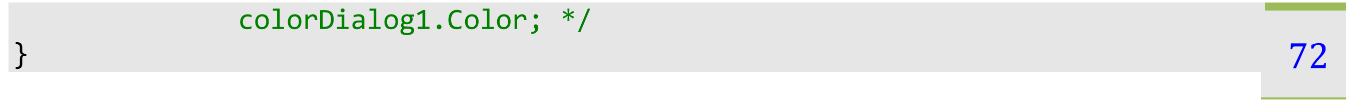
Lastnost	Razlaga
AllowFullOpen	Omogočen/onemogočen gumb za mešanje barv v oknu.
AnyColor	Če lastnost postavljena na *True* bodo v množici bazičnih barv prikazane vse možne barve.
Color	Predzbrana barva gradnika.
SolidColorOnly	Barve v pogovornem oknu bodo/ne bodo omejene le na prave/pristne barve (osnovne barve brez senc in poltonov) .

Tabela 12: Najpomembnejše lastnosti pogovornega okna *ColorDialog*.

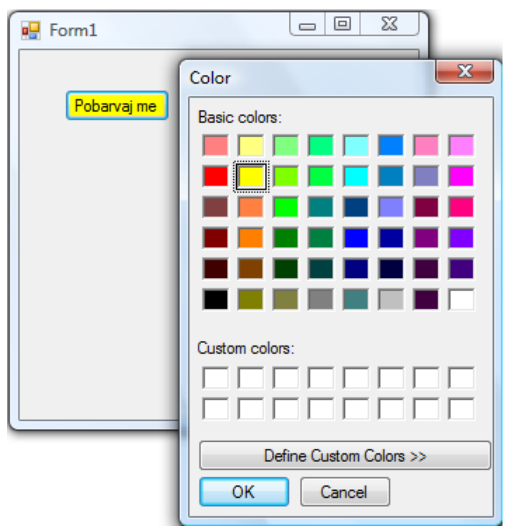
Naslednji primer prikazuje, kaj se zgodi ob dogodku *Click* gumba z imenom *bPobarvaj*. Ob kliku na ta gumb se je odprlo pogovorno okno *ColorDialog* in če uporabnik v tem oknu izbere poljubno barvo in okno zapre s klikom na gumb *OK* (*V redu*), se gumb pobarva z izbrano barvo.

To dosežemo z naslednjo odzivno metodo.

```
private void bPobarvaj_Click(object sender, EventArgs e)
{
    if (colorDialog1.ShowDialog()==DialogResult.OK)
            (sender as Button).BackColor = colorDialog1.Color;
            /*lahko pa bi zapisali tudi takole:  bPobarvaj.BackColor =
```

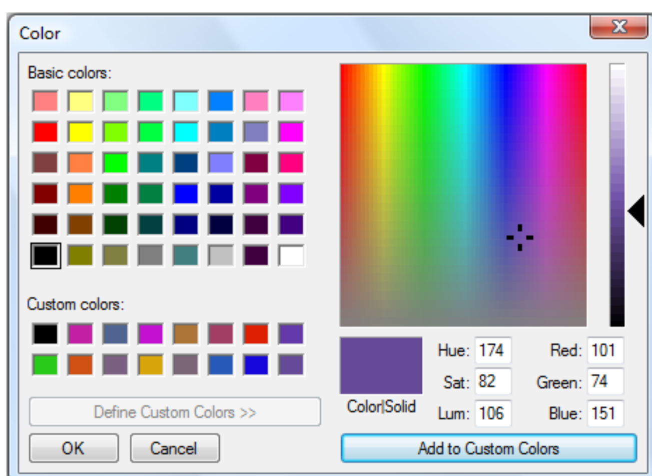


```
                colorDialog1.Color; */
}
```



Slika 44: Pogovorno okno *ColorDialog*.

Posebnost okna za izbiranje barv je njegova tabelarična lastnost *CustomColors* (barve po meri), kamor lahko uporabnik med izvajanjem programa shrani 16 svojih, iz osnovnih barv narejenih oz. "namešanih" barv. Odpremo ga s klikom na gumb *Define Custom Colors*. Okno se razširi in v razširjenem oknu lahko z dvoklikom na barvni paleti najprej izberemo ustrezno barvo, nato pa jo s klikom na *Add to Custom Colors* dodamo na paleto *Custom Colors*. Izbrane barve so v tej paleti shranjene ves čas delovanja programa.



Slika 45: Ustvarjanje lastne barve palete v pogovornem oknu *ColorDialog*.

Vsak dialog (vsako pogovorno okno) lahko definiramo in ga prikažemo tudi povsem programsko, ne da bi prej na obrazec postavili ustrezen gradnik. To dosežemo z dinamičnim zaseganjem pomnilnika (ustvarjanjem novega objekta ustreznega razreda) za konkreten dialog. Za *ColorDialog* bi to naredili npr. takole:

```
private void button1_Click_1(object sender, EventArgs e)
```

```
{
   //najprej ustvarimo nov objekt tipa ColorDialog. Ime objekta naj bo cD1
   ColorDialog cD1=new ColorDialog();
   /*dialog odpremo z metodo ShowDialog in preverimo če je uporabnik
     kliknil gumb V redu*/
   if (cD1.ShowDialog()==DialogResult.OK)
      (sender as Button).BackColor = cD1.Color;
      /*Če je bil kliknjen gumb V redu, pobarvamo pošiljatelja (v našem
        primeru gumb button1). Parameter sender smo uporabili tudi v programu
        Hitrostno klikanje*/
}
```

Naslednji primer prikazuje, kako dinamično ustvarimo nov barvni dialog in nato z njegovo pomočjo določimo barvo gradnika *TextBox* (ime gradnika *textBox1*) – kodo zapišemo npr. ob dogodku *Click* gumba *button2*.

```
private void button2_Click(object sender, EventArgs e)
{
    ColorDialog MyDialog = new ColorDialog();//Ustvarimo nov dialog
    MyDialog.AllowFullOpen = false; // Onemogočimo gumb za mešanje barv
    MyDialog.ShowHelp = true; // Uporabniku omogočimo dostop za gumb Pomoč.
    /*Dialogu ColorDialog nastavimo začetno barvo na trenutno barvo
      gradnika textBox1.*/
    MyDialog.Color = textBox1.ForeColor;
    /*Če uporabnik v ColorDialog klikne gumb V redu, spremenimo barvo v
      gradniku textBox1 */
    if (MyDialog.ShowDialog() == DialogResult.OK)
        textBox1.ForeColor = MyDialog.Color;
}
```

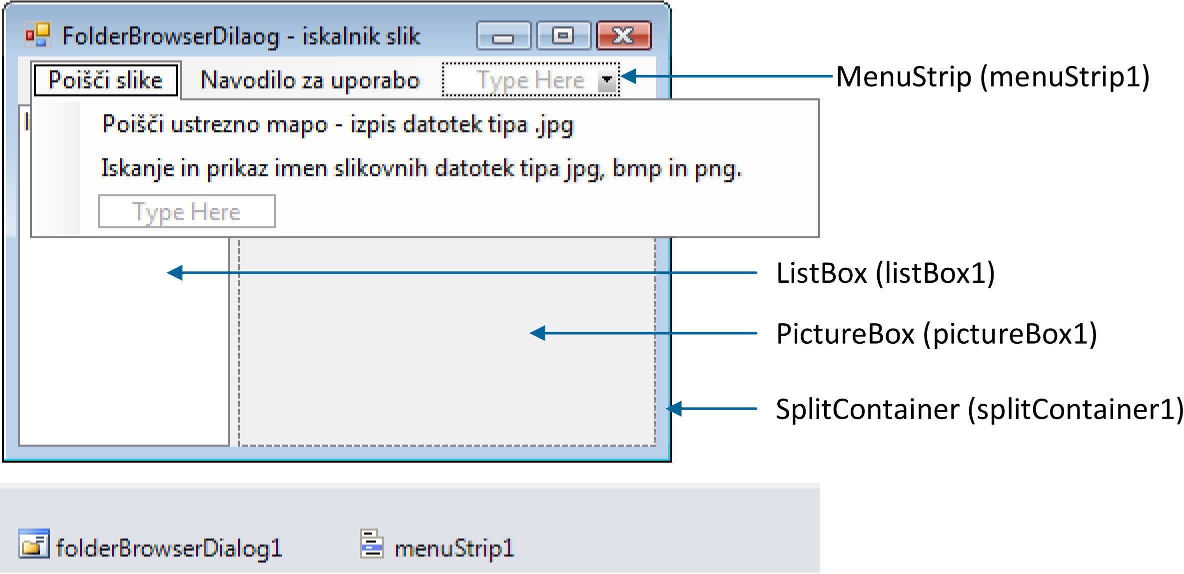
FolderBrowserDialog – pogovorno okno za raziskovanje map

Pogovorno okno *FolderBrowserDialog* omogoča uporabniku premikanje po mapah računalnika in ustvarjanje novih map. Dialog je nekakšen komplement dialogu *OpenFileDialog*, ki pa se uporablja za raziskovanje in določanje imen datotek. Okno odpremo z metodo *ShowDialog*, omogoča pa izbiro poljubne mape. Izbiro potrdimo s klikom na gumb *OK* (*V redu*). To pomeni, da okno vrne vrednost *DialogResult.OK.*

Lastnost	Razlaga
Description	Nastavitev poljubnega teksta (naslova), ki se izpiše nad drevesno strukturo map v oknu.
SelectedPath	Če že pred odpiranjem dialoga tu vnesemo vrednost, s tem določimo mapo, ki bo izbrana, ko se bo dialog odprl. Ob zaključku pa ta lastnost vsebuje ime izbrane mape.
ShowNewFolderButton	V oknu je privzeto tudi gumb, ki omogoča ustvarjanje novih map. Če lastnost *ShowNewFolderButton* postavimo na *false*, tega gumba ni.

Tabela 13: Glavne lastnosti pogovornega okna *FolderBrowserDialog*.

Uporabo *FolderBrowserDialog* dialoga prikažimo na primeru preprostega iskalnika slik. Na obrazec postavimo gradnik *MenuStrip* in v njem ustvarimo postavke kot jih prikazuje spodnja slika. Dodajmo še *SplitContainer* – ta gradnik vsebuje dve plošči (dva panela), ki ju lahko med izvajanjem poljubno širimo in ožimo. Na levi panel nato postavimo gradnik *ListBox* (*Dock = Fill*), v desnega pa *PictureBox* (*Dock* = Fill, *SizeMode=StretchImage*). Gradnik *ListBox* že s svojim imenom nakazuje, da se uporablja za izpisovanje seznamov, rezultatov poizvedb, ipd. Na obrazec postavimo še gradnik *FolderBrowserDialog*, s pomočjo katerega bomo iskali mapo s slikami.



Slika 46: Gradniki na obrazcu za prikaz pogovornega okna *FolderBrowserDialog*.

Uporabili smo gradnik *PictureBox*. Namen tega gradnika je prikaz poljubne slike.

S pomočjo pogovornega okna *FolderBrowserDialog* in dogodkov *Click* glavnega menija, bomo v oknu *ListBox* prikazali enkrat imena vseh slikovnih datotek tipa *jpg*, v drugem primeru pa slikovnih datotek tipov *jpg*, *bmp* in *png*. Uporabili bomo metodo *GetFiles* razreda *Directory*. Posredujemo ji dva parametra: mapo, ki jo preiskujemo in tipa datotek, ki jih iščemo v tej mapi. Metoda vrne seznam vseh datotek, ki ga priredimo tabeli nizov.

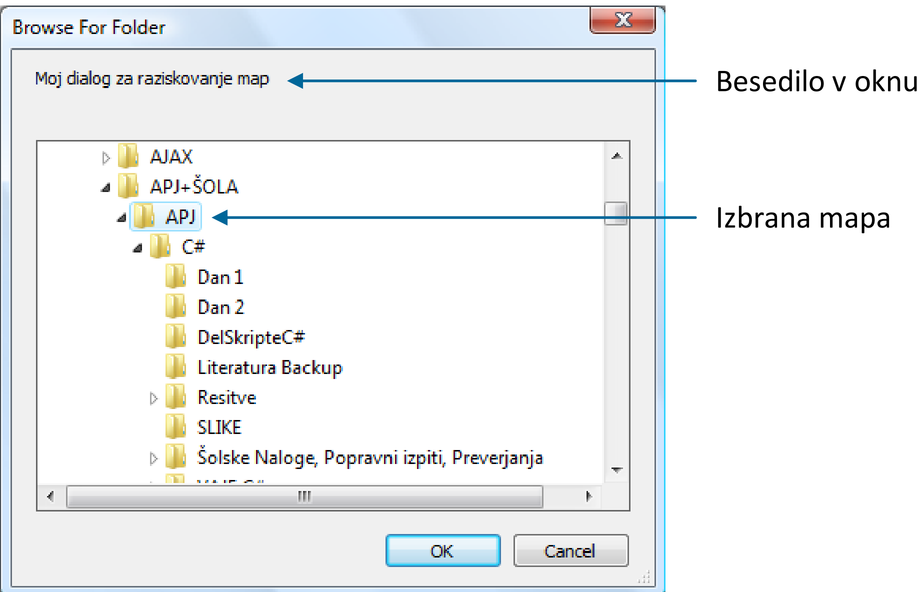
```
private void SlikeJPG_Click(object sender, EventArgs e)
{
    if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
    {
        string[] fileArray;
        //imena vseh datotek tipa jpg izbrane mape zapišemo v tabelo nizov
        fileArray = Directory.GetFiles(folderBrowserDialog1.SelectedPath,
"*.jpg");
         foreach (string imeDat in fileArray)//slike dodamo v gradnik ListBox
            listBox1.Items.Add(imeDat);
    }
}
private void VseSlike_Click(object sender, EventArgs e)
{
    if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
```

```
    {
        string[] fileArray;
        //imena vseh datotek izbrane mape zapišemo v tabelo nizov
        fileArray = Directory.GetFiles(folderBrowserDialog1.SelectedPath,
"*.*");
        foreach (string imeDat in fileArray) //obdelamo tabelo
        {
            //črke v imenih datoteke spremenimo v velike črke
            string ime = imeDat.ToUpper();
            //če gre za datoteko s ustrezno končnico, jo dodamo v seznam
            if (ime.EndsWith(".JPG") || ime.EndsWith(".BMP") ||
ime.EndsWith(".PNG"))
                listBox1.Items.Add(imeDat);
        }
    }
}
private void NavodiloZaUporabo_Click(object sender, EventArgs e)
{
    MessageBox.Show("V meniju 'Poišči slike' najprej poišči mapo s
slikami.\nSeznam slikovnih datotek se nato prikaže v gradniku ListBox.\n\nOb
kliku na vrstico ListBox-a se prikaže ustrezna slika!\n\nMejo med seznamom
slik in sliko lahko premikaš levo-desno.\n\nCeloten obrazec lahko poljubno
povečaš ali zmanjšaš.", "Navodilo za uporabo", MessageBoxButtons.OK,
MessageBoxIcon.Information);
}
private void listBox1_Click(object sender, EventArgs e)
{
    try
    {
        pictureBox1.Image = Image.FromFile(listBox1.SelectedItem.ToString());
    }
    catch
    { }
}
```

Oglejmo si še primer dinamičnega ustvarjanja *FolderBrowserDialog*-a in programske spremembe nekaterih nastavitev. Dialog smo poimenovali *MojDialog*.

```
private void button3_Click(object sender, EventArgs e)
{
    FolderBrowserDialog MojDialog = new FolderBrowserDialog();
    //Določimo tekst v oknu, tik nad prikazom map
    MojDialog.Description = "Moj dialog za raziskovanje map";
    //Določimo privzeto ime mape, ki bo izbrana ob prikazu dialoga
    MojDialog.SelectedPath = "c:\\Program Files\\Common Files";
    //Onemogočimo gumb za ustvarjanje novih map
    MojDialog.ShowNewFolderButton = false;

    if (MojDialog.ShowDialog() == DialogResult.OK)
    {    /*stavki, ki naj se izvedejo če uporabnik izbiro mape potrdi s
         klikom na gumb OK (V redu)*/
    }
}
```

Pri navajanju privzete poti v lastnosti *SelectedPath* moramo zapisati znaka \\ , ali pa pred začetkom niza zapisati znak @. Že od prej namreč vemo, da enojni znak \ v nizu pomeni ubežni niz.



Slika 47: *FolderBrowserDialog* z nekaterimi programskimi prednastavitvami.

FontDialog – pogovorno okno za izbiro pisave.

Pogovorno okno *FontDialog* služi za izbiro pisave. Če uporabnik pisavo izbere in okno zapre s klikom na gumb *V redu***,** okno vrne vrednost *DialogResult.OK*, izbrana pisava pa se shrani v lastnost okna *Font*.

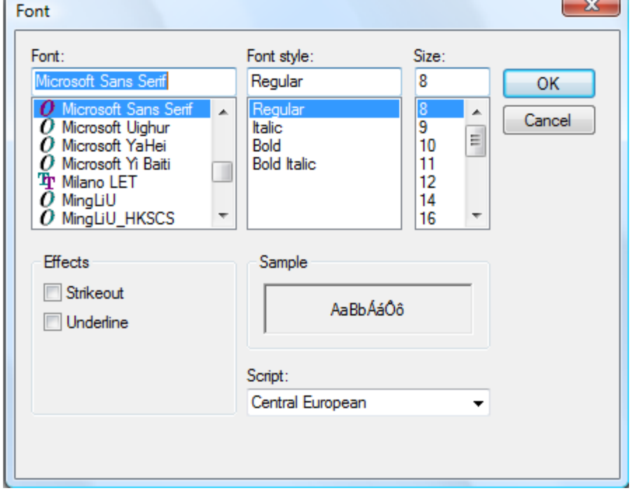
Naslednji primer pa prikazuje, kako s pomočjo dialoga za izbiro pisave spremenimo pisavo gradnika *TextBox* (ime gradnika je *textBox1*). Prikazana sta dva načina: v prvem primeru upoštevamo vse uporabnikove nastavitve v oknu *FintDialog*, v drugem (v obliki komentarja) pa le nekatere (ime, velikost in stil pisave). Poudarimo, da s tem le določimo ime datoteke. Same datoteke pa s tem še ne odpremo. Nekatere lastnosti pogovornega okna *FontDialog* so zbrane v naslednji tabeli:

Lastnost	Razlaga
Color	Izbrana barva pisave.
Font	Izbrana vrste pisave.
MaxSize	Nastavitev največje velikosti pisave, ki jo uporabnik še lahko izbere.
MinSize	Nastavitev najmanjše velikosti pisave, ki jo uporabnik še lahko izbere.
ShowApply	V oknu bo oz. ne bo gumba *Uporabi*.
ShowColor	V oknu bo oz ne bo možno izbiranje barve pisave.
ShowEfects	V oknu bo oz. ne bo prikazana postavka *Učinki* (prečrtanje, podčrtavanje in izbira barve).

ShowHelp	V oknu bo oz. ne bo gumba *Pomoč*.

Tabela 14: Najpomembnejše lastnosti pogovornega okna *FontDialog*.

```
if (fontDialog1.ShowDialog() == DialogResult.OK)
{
     textBox1.Font = fontDialog1.Font;
     /*lahko tudi textBox1.Font = new Font(fontDialog1.Font.Name,
       fontDialog1.Font.Size ,fontDialog1.Font.Style);*/
}
```



Slika 48: Pogovorno okno *FontDialog* za izbiro pisave.

OpenFileDialog - pogovorno okno za odpiranje datotek

Pogovorno okno *OpenFileDialog* je namenjeno izbiri datoteke, ki jo nameravamo odpreti. Okno odpremo z metodo *ShowDialog*. Če uporabnik datoteko izbere in okno zapre s klikom na gumb *OK* (*V redu*), okno vrne vrednost *DialogResult.OK*, ime izbrane datoteke pa se shrani v lastnost okna *FileName*.

Lastnost	Razlaga
AddExtension	Lastnost določa, ali naj pogovorno okno avtomatično doda datoteki končnico, v primeru, da jo je uporabnik pozabil napisati.
CheckFileExists	Lastnost določa, ali naj pogovorno okno izpiše obvestilo v primeru, da je uporabnik navedel ime datoteke, ki ne obstaja.
CheckPathExists	Lastnost določa, ali naj pogovorno okno izpiše obvestilo v primeru, da je uporabnik navedel pot do datoteke, ki ne obstaja.
DefaultExt	Nastavitev privzete končnice datoteke.
FileName	Ime datoteke, ki naj bo izbrana v pogovornem oknu, po izhodu pa tu dobimo "rezultat".
Filter	Omejitev le na določene vrste datotek (npr. tekstovne, ..).
InitialDirectory	Nastavitev privzetega imenika.

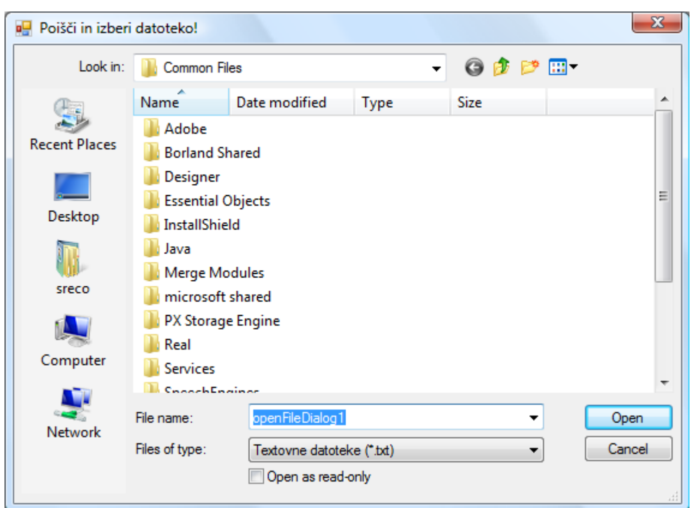
Multiselect	Lastnost določa, ali lahko v pogovornem oknu hkrati izberemo več datotek
ReadOnlyChecked	Lastnost določa kakšna je privzeta nastavitev za možnost *ReadOnly*, ki uporabnika opozarja, da izbrano datoteko lahko odpre le branje. Nastavitev opcije je smiselna le v primeru, da je lastnost *ShowReadOnly* nastavljena na *True*.
ShowReadOnly	Lastnost določa, ali naj se v pogovornem okno prikaže stikalo *Samo za branje*.
Title	Naslov pogovornega okna.

Tabela 15: Lastnosti pogovornega okna *OpenFileDialog*.

Naslednji primer prikazuje, kako s pomočjo *OpenFileDialoga* ime neke datoteke zapišemo v polje *TextBox* (ime gradnika je *textBox1*).

```
/*Pred odpiranjem okna lahko nastavimo tudi filter, s katerim povemo kakšne
vrste datoteke želimo prikazati v oknu*/
openFileDialog1.Filter = "Textovne datoteke (*.txt)|*.txt|Moje datoteke
(*.MOJ)|*.MOJ";

/*s stikalom Read Only lahko uporabnika obvestimo, da bo izbrano datoteko
lahko le pregledoval, ne pa tudi ažuriral*/
openFileDialog1.ShowReadOnly = true;

//POZOR: dva znaka \\ za dostop do podmape pišemo zaradi tolmačenja znaka \ v
//nizu
openFileDialog1.InitialDirectory = "c:\\Program Files\\Common
Files";//Privzeti imenik

//Določimo lahko tudi naslov pogovornega okna
openFileDialog1.Title = "Poišči in izberi datoteko!";

//Pogovorno okno odpremo z metodo ShowDialog
if (openFileDialog1.ShowDialog() == DialogResult.OK) textBox1.Text =
openFileDialog1.FileName;
```

V prikazanem primeru smo nastavili tudi lastnost *Filter*, s katero povemo, katere vrste datotek bodo v oknu prikazane. Z ločilno črto "|" delimo filter v dva dela: prvi del pomeni besedilo v spustnem seznamu za izbiro tipa datoteke, drugi del pa so dejanski tipi datotek. Nastavimo lahko več filtrov hkrati. Z lastnostjo *ShowReadOnly* lahko uporabnika tudi obvestimo, da datotek ne bo mogel spreminjati. Z lastnostjo *InitialDirectory* smo dosegli, da se pri odpiranju dialoga prikaže vsebina točno določene mape. Lastnost *Title* določa napis na vrhu okna *OpenFileDialog* (privzeti zapis bo sicer *Open*).



Slika 49: Pogovorno okno *OpenFileDialog*.

SaveFileDialog - pogovorno okno za shranjevanje datotek

Pogovorno okno *SaveFileDialog* je namenjeno pomoči pri shranjevanju datotek. Če uporabnik datoteko izbere in okno zapre s klikom na gumb *OK* (*V redu*)**,** okno vrne vrednost *DialogResult.OK*, ime izbrane datoteke pa se shrani v lastnost okna *FileName*.

S pogovornima oknoma *OpenFileDialog* in *SaveFileDialog* datoteke ni mogoče ne odpreti ne shraniti. Pogovorni okni samo omogočata izbiro datoteke, ki jo nameravamo odpreti oziroma shraniti. Za odpiranje in zapiranje datoteke moramo poskrbeti z ustrezno programsko kodo.

Lastnost	Razlaga
AddExtension	Lastnost določa, ali naj se datoteki, v primeru da jo je uporabnik pozabil napisati, avtomatično doda končnica. Če je vrednost nastavljena na *True*, se imenu datoteke (seveda le v primeru, da lastnost filter ni nastavljena), avtomatično doda končnica, zapisna v *DefaultExt*.
CheckFileExists	Lastnost določa, ali naj se izpiše obvestilo v primeru, da je uporabnik navedel ime datoteke, ki ne obstaja.
CheckPathExists	Lastnost določa, ali naj se izpiše obvestilo v primeru, da je uporabnik navedel pot do datoteke, ki ne obstaja.
CreatePrompt	Nastavitev dovoljenja za ustvarjanje datoteke, če uporabnik navede ime datoteke, ki še ne obstaja.
DefaultExt	Nastavitev privzete končnice datoteke.
FileName	Ime datoteke, ki jo izberemo v pogovornem oknu, oz. ime datoteke, ki smo ga zapisali sami. Po zapiranju okna tu dobimo "rezultat".
Filter	Omejitev le na določene vrste datotek (npr. tekstovne, ..).
InitialDirectoty	Nastavitev privzetega imenika.

OverwritePrompt	Lastnost določa, ali naj nas program opozori, če za ciljno datoteko izberemo datoteko, ki že obstaja.
Title	Naslov pogovornega okna.

Tabela 16: Tabela lastnosti pogovornega okna *SaveFileDialog*.

V naslednjem primeru bo prikazana še uporaba gradnika *RichTextBox*. Besedilo, ki ga uporabnik zapiše v ta gradnik, je lahko poljubno oblikovano. Primer prikazuje, kako v datoteko zapišemo celotno vsebino gradnika *RichTextBox* (ime gradnika je *richTextBox1*). Pri tem si za izbiro datoteke pomagamo z gradnikom *SaveFileDialog*.

```
private void button2_Click(object sender, EventArgs e)
{
    saveFileDialog1.OverwritePrompt = true; /*program naj nas opozori, če
                                              datoteka že obstaja*/
    //Določimo privzeti imenik
    saveFileDialog1.InitialDirectory = "c:\\Programi";
    saveFileDialog1.Title = "Shrani datoteko!"; //Naslov pogovornega okna
    //Nastavitev filtra
    saveFileDialog1.Filter = " Moje rtf datoteke (*.rtf)|*.rtf";
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
        richTextBox1.SaveFile(saveFileDialog1.FileName,
RichTextBoxStreamType.RichText);
}
```

V prikazanem primeru smo s pomočjo lastnosti *OverwritePrompt* najprej poskrbeli, da nas program pri shranjevanju v datoteko opozori, če ta že obstaja. Lastnost *Title* določa napis na vrhu okna *SaveFileDialog* (privzeti zapis bo sicer *Save As*).

Predvajalnik glasbe in videa

Visual C# zna delati tudi z glasbenimi in video datotekami. Ustvarimo nov projekt in ga poimenujmo *Glasba*. Spoznali bomo razred *SoundPlayer* za delo z glasbenimi datotekami in gradnik *WindowsMediaPlayer* za predvajanje glasbe in videa.

Na obrazec postavimo gradnik *MenuStrip* z izbirami:

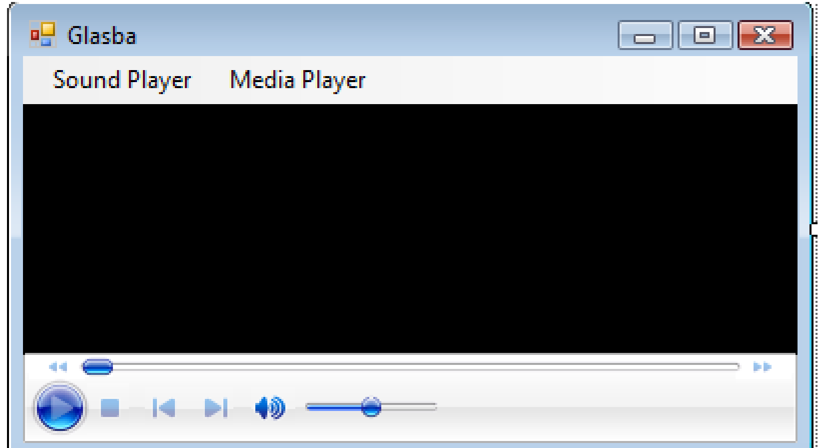
- SoundPlayer
    - Zaigraj privzeto skladbo
    - Zaigraj mojo skladbo
    - Poišči glasbo
    - Zaključi predvajanje
- MediaPlayer
    - Moja glasba
    - Poišči glasbo
    - Video

Za prvo postavko menija bomo potrebovali razred *SoundPlayer* (objekti tipa *SoundPlayer* so namenjeni predvajanju glasbe tipa *wav*). V sekciji *using* moramo dodati stavek

```
using System.Media;
```

Drugo postavko menija *SoundPlayer* bomo realizirali s pomočjo gradnika *WindowsMediaPlayer*. V okno *ToolBox* dodajmo že pripravljen gradnik *Windows Media Player* takole: preklopimo na pogled *Design* → *ToolBox* → *General* → desni klik miške → *Choose Items* → *COM Components*, poiščemo *WindowsMediaPlayer*, ga odkljukamo in izbiro potrdimo s klikom na gumb *OK*. V paleti *General* se pojavi nov gradnik *Windows Media Player*, ki ga postavimo na obrazec. Gradniku nastavimo še lastnost *Dock=Fill*.

Prvo postavko menija *SoundPlayer* (*Zaigraj privzeto skladbo*) izvedemo tako, da *Solution Explorerju* izberimo *Glasba→Properties*. V oknu, ki se prikaže, izberimo *Resources→Audio* in s potegom miške (*Drag & Drop*) v to okno povlecimo poljubno datoteko tipa .wav (v našem primeru je to datoteka *AmericanPie.WAV*).



Slika 50: Obrazec za predvajanje glasbe in videa.

Druga postavka menija *SoundPlayer* je podobna prvi, le da smo datoteko tipa *.wav* naprej prekopirali v točno določeno mapo diska C, od koder jo nato predvajamo. S prvo in drugo postavko menija smo tako le prikazali, kako lahko določeno skladbo (ali pa seveda zvok) vključimo v projekt na dva načina. Na računalniku moramo v ta namen najprej poiskati dve glasbeni datoteki in potem ustrezno nastaviti vrednost spremenljivki *PotDoGlasbe*. Pri takih projektih bi bilo torej smiselno programiranje ustreznih nastavitev naše aplikacije. Podatke o poteh do podatkov (datotek, slik, zvokov, …) bi zapisali v posebno datoteko, nastavili pa bi jih pri izdelavi namestitvenega programa.

Tretja postavka menija *SoundPlayer* prikazuje, kako lahko neko skladbo poiščemo s pomočjo dialoga *OpenFileDialog*.

Prva postavka menija *MediaPlayer* prikazuje, kako lahko glasbeno datoteko, ki se nahaja v določeni mapi, predvajamo s pomočjo *MediaPlayer*-ja, druga in tretja postavka pa, kako lahko glasbeno oz. video datoteko poiščemo s pomočjo objekta tipa *OpenFileDialog*. Ime izbrane datoteke moramo zapisati v lastnost URL našega predvajalnika.

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

```
    //V sekcijo using najprej vključimo stavek using System.Media;
    //objekti, ustvarjeni iz razreda SoundPlayer, igrajo le glasbo tipa wav!
    public SoundPlayer mojPredvajalnikGlasbe = new SoundPlayer();
    public Form1()
    {
        InitializeComponent();
    }
    private void ZaigrajPrivzetoSkladboToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        /*V Solution Explorerju izberimo našo rešitev (Glasba)->Properties.
          V oknu, ki se prikaže, izberimo Resources-->Audio. S potegom
          miške (Drag & Drop) nato v to okno povlecimo poljubno datoteko
          tipa .wav. Okno Properties nato lahko zapremo.*/
          mojPredvajalnikGlasbe.Stream = Properties.Resources.American_Pie;
          //z metodo PlayLooping začnemo predvajanje, ko pa se skladba konča
          //se predvajanje avtomatično prične znova
          mojPredvajalnikGlasbe.PlayLooping();
    }
    private void ZaigrajMojoSkladboMapiToolStripMenuItem_Click(object sender,
EventArgs e)
    {
         string potDoGlasbe = @"C:\Glasba\Lolita.wav";
         mojPredvajalnikGlasbe.SoundLocation = potDoGlasbe;
         mojPredvajalnikGlasbe.Play();
    }
    private void PoiščiGlasboToolStripMenuItem_Click(object sender, EventArgs
e)
    {
        OpenFileDialog opf = new OpenFileDialog();
        opf.Filter = "Glasbene datoteke|*.wav";
        if (opf.ShowDialog() == DialogResult.OK)
        {
            mojPredvajalnikGlasbe.SoundLocation = opf.FileName;
            mojPredvajalnikGlasbe.Play();
        }
    }
    private void ZaključiPredvajanjeToolStripMenuItem_Click(object sender,
EventArgs e)
    {
        //zaustavitev predvajalnika
        mojPredvajalnikGlasbe.Stop();
    }
    private void MojaGlasba_Click(object sender, EventArgs e)
    {
        //določimo pot in ime skladbe za predvajanje v MediaPlayer-ju
        string potDoGlasbe = @"C:\Glasba\Krompir.mp3";
        WindowsMediaPlayer.settings.autoStart = true;
        WindowsMediaPlayer.URL = potDoGlasbe;
        WindowsMediaPlayer.Visible = true;
    }
    private void PoiščiGlasboToolStripMenuItem2_Click(object sender,
EventArgs e)
```

```
    {
        /*s pomočjo OpenFileDialoga poiščemo skladbo za predvajanje v
          MediaPlayer-ju*/
        OpenFileDialog opf = new OpenFileDialog();
        //dovolimo iskanje le glasbenih datotek tipa wav in mp3
        opf.Filter = "Glasbene datoteke(*.wav *.mp3)|*.wav;*.mp3";
        if (opf.ShowDialog() == DialogResult.OK)
        {
            WindowsMediaPlayer.settings.autoStart = true;
            // izbrano glasbeno datoteko zapišemo v lastnost URL
            WindowsMediaPlayer.URL = opf.FileName;
            WindowsMediaPlayer.Visible = true;
        }
    }
    private void VideoToolStripMenuItem_Click(object sender, EventArgs e)
    {
        OpenFileDialog opf = new OpenFileDialog();
        //dovolimo iskanje le video datotek tipa WMV in AVI
        opf.Filter = "Video datoteke(*.WMV *.AVI)|*.WMV;*.AVI";
        if (opf.ShowDialog() == DialogResult.OK)
        {
            WindowsMediaPlayer.settings.autoStart = true;
            //izbrano video datoteko zapišemo v lastnost URL
            WindowsMediaPlayer.URL = opf.FileName;
            WindowsMediaPlayer.Visible = true;
        }
    }
}
```

Preprosti urejevalnik besedil

Izdelajmo svoj preprosti urejevalnik besedil. Besedilo bomo lahko urejali in oblikovali, shranjevali v tekstovne datoteke, možno pa bo tudi odpiranje že obstoječih tekstovnih datotek.

Na obrazec zaporedoma postavimo gradnike *MenuStrip*, *ToolBox*, *StatusStrip*. V srednji del postavimo še *Panel* (*Dock=Fill*), nanj pa gradnik *TextBox* (*Dock=Fill*). Glavni meni oblikujmo tako, da bo vseboval opcije:

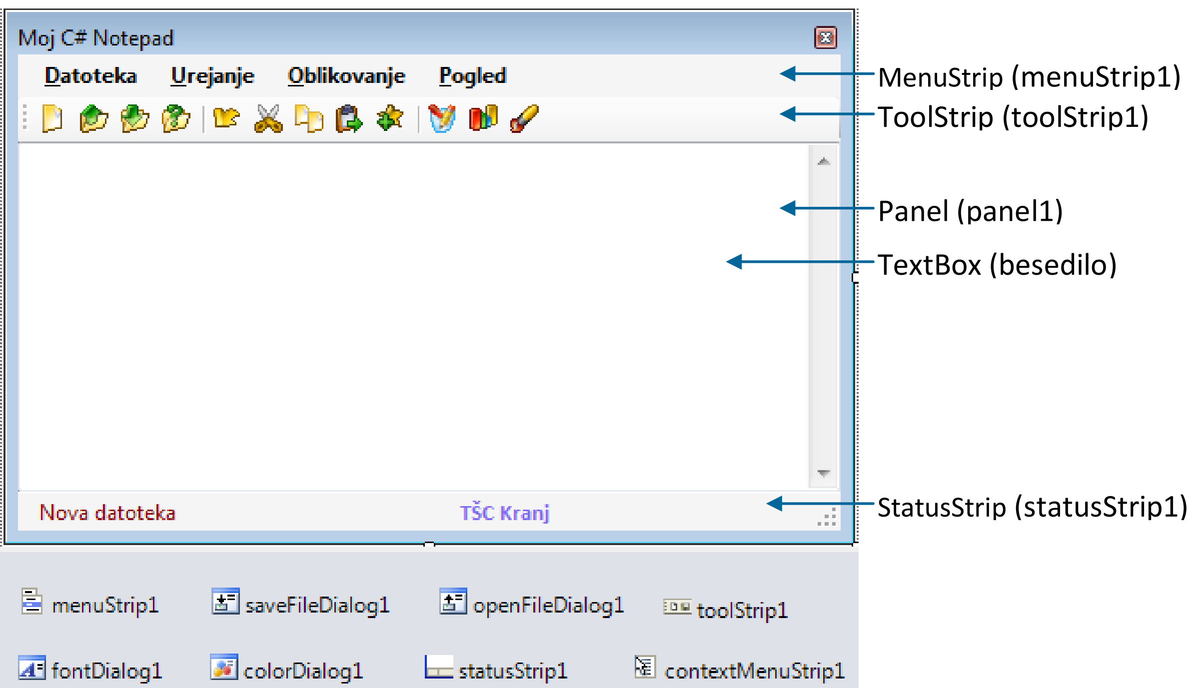
- Datoteka
  - Nova
  - Odpri
  - Shrani
  - Shrani kot…
  - Izhod
- Urejanje
  - Razveljavi
  - Izreži
  - Kopiraj
  - Prilepi

  - Izberi vse
- Oblikovanje
  - Pisava
  - Barva pisave
  - Poravnava besedila
    - Levo
    - Sredinsko
    - Desno
  - Ozadje
- Pogled
  - Statusna vrstica

V orodjarno dodajmo 12 gumbov in jim priredimo primerne slike tipa *bmp*. Eden od možnih virov za sličice je npr. *http://qicons.cubeactive.com/*.

Spomnimo se, da gumbe v orodjarno dodajamo z desnim klikom na gradnik tipa *ToolStrip*, za katerim si izberemo *Button*. Dogodki, ki jih bomo priredili gumbom v orodjarni, se bodo ujemali z vrstnim redom postavk glavnega menija.

Naredimo še dva predalčka v gradniku *StatusStrip*. Prvemu določimo lastnost *AutoSize* na *False*, nato pa spremenimo *Size*→*Width* na 100. S tem bomo dosegli zadostno širino za tekst, ki ga želimo v predalčku prikazati. Drugemu nastavimo lastnost *Spring* na *True*, da se razširi čez celotno preostalo širino gradnika *StatusStrip*). Prvi predalček bo hranil ime datoteke, ki jo trenutno urejamo. Če pa datoteka še ni shranjena, bo v njem napis *Nova datoteka* (takšno je tudi začetno besedilo).



Slika 51: Gradniki *Visual C#* urejevalnika.

V projekt dodajmo še lebdeči meni (*ContextMenuStrip*) z naslednjimi opcijami:

- Izreži
- Kopiraj
- Prilepi
- Izberi vse
- Razveljavi
- Pisava
- Barva pisave
- Ozadje

Imena dogodkov se ujemajo z njihovim namenom. Zato v spodnji kodi ne bo težko ugotoviti, kateri opciji glavnega menija, lebdečega menija ali pa gumbom v orodjarni pripadajo.

```
//Oznaka, da gre za novo datoteko, ki še ni shranjena
private string nova = "Nova datoteka";
public Form1()//Konstruktor obrazca
{
    InitializeComponent();
    //oznaka, da gre za novo, še neimenovano datoteko
    toolStripStatusLabel1.Text=nova;
}
//Nova datoteka
private void novaToolStripMenuItem_Click(object sender, EventArgs e)//Nova
datoteka
{
    ShraniSpremembe("Nov dokument");
    besedilo.Clear();
    toolStripStatusLabel1.Text = nova;
}
//metoda za preverjanje sprememb
private void ShraniSpremembe(string napis)
{
    //preverimo, če je besedilo v urejevalniku spremenjeno in še neshranjeno
    if (besedilo.Modified)
    {
        if (MessageBox.Show("Shranim spremembe?", napis,
MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            /*če je uporabnikova odločitev DA,se vsebina gradnika besedilo
              shrani v datoteko*/
            if (toolStripStatusLabel1.Text != nova)
            {
                try
                {
                    //zapis v datoteko
                    File.WriteAllText(toolStripStatusLabel1.Text,
besedilo.Text);
                }
                catch
                { MessageBox.Show("Napaka pri pisanju v datoteko!"); }
```

```
            }
            else VDatoteko();//klic metode za shranjevanje v datoteko
        }
    }
}
//metoda za shranjevanje v datoteko
private void VDatoteko()
{
    try
    {
        if (saveFileDialog1.ShowDialog() == DialogResult.OK)
        {
            //preverimo, če datoteka s tem imenom že obstaja
            if (File.Exists(saveFileDialog1.FileName))
            {
                if (MessageBox.Show("Datoteka s tem imenom že obstaja! Ali 
naj jo prepišem?", "POZOR!", MessageBoxButtons.YesNo) == DialogResult.Yes)
                {
                    File.WriteAllText(saveFileDialog1.FileName,
                                      besedilo.Text);
                    toolStripStatusLabel1.Text = saveFileDialog1.FileName;
                }
            }
            else
            {
                File.WriteAllText(saveFileDialog1.FileName, besedilo.Text);
                toolStripStatusLabel1.Text = saveFileDialog1.FileName;
            }
        }
    }
    catch
    { MessageBox.Show("Napaka pri pisanju v datoteko!"); }
}
```

Pokazali smo, kaj vse je potrebno postoriti pred odpiranjem nove datoteke. Za preverjanje sprememb in shranjevanje v datoteko smo napisali dve novi metodi. Sledi pa nekaj odzivnih metod za odpiranje že obstoječe datoteke in prenos vsebine te datoteke v gradnik *besedilo*, ter za shranjevanje oz. zaključek programa.

```
//Odpri datoteko
private void odpriToolStripMenuItem_Click(object sender, EventArgs e)
{
    ShraniSpremembe("Shranjevanje");
    besedilo.Clear();
    toolStripStatusLabel1.Text = nova;
    //datoteko bo poiskal oz. vpisal ime uporabnik
    openFileDialog1.FileName = "";
    try //odpiranje nove datoteke
    {
        if (openFileDialog1.ShowDialog() == DialogResult.OK)
        {
            string myString = File.ReadAllText(openFileDialog1.FileName);
            besedilo.Text = myString;
```

```
            toolStripStatusLabel1.Text = openFileDialog1.FileName;
        }
    }
    catch {  }
}
//Shrani datoteko
private void shraniToolStripMenuItem_Click(object sender, EventArgs e)
{
    ShraniSpremembe("Shranjevanje");
}
//Shrani kot
private void toolStripMenuItem3_Click(object sender, EventArgs e)
{
    VDatoteko(); //shranjevanje v datoteko
}
//Izhod
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    ShraniSpremembe("Izhod");
    Application.Exit();//Zaključek programa
}
```

Urejevalnik bo vseboval tudi nekatere možnosti, ki jih vsebuje večina urejevalnikov (razveljavi, izreži, kopiraj, prilepi). Pri kopiranju in lepljenju si pomagamo z metodami razreda *ClipBoard*, ki nam pomagajo pri pridobivanju in shranjevanju podatkov na odložišče sistema *Windows* (začasno odlagališče podatkov, kamor sistem *Windows* shranjuje vse, kar smo označili in za to izbrali ukaz *Kopiraj* ali *Izreži*).

```
//Undo
private void razveljavi_Click(object sender, EventArgs e)
{
    /*preverim, če je razveljavitev možna: v tem primeru, ima lastnost
        CanUndo gradnika TextBox vrednost True*/
    if (besedilo.CanUndo == true)
        besedilo.Undo();
}
//Izreži
private void izrezi_Click(object sender, EventArgs e)
{
    Clipboard.Clear();//brisanje vsebine odložišča
    /*če izbrano besedilo vsebuje vsaj en znak (lastnost SelectionLength
    je večja od 0), ga shranim v odložišče*/
    if (besedilo.SelectionLength > 0)
        //lastnost SelectedText označuje izbrano besedilo gradnika TextBox
        Clipboard.SetText(besedilo.SelectedText);
    besedilo.SelectedText = "";
}
//Kopiraj
private void kopiraj_Click(object sender, EventArgs e)
{
    Clipboard.Clear(); //čiščenje odložišča
    if (besedilo.SelectionLength > 0)//če je izbran poljubnen tekst
```

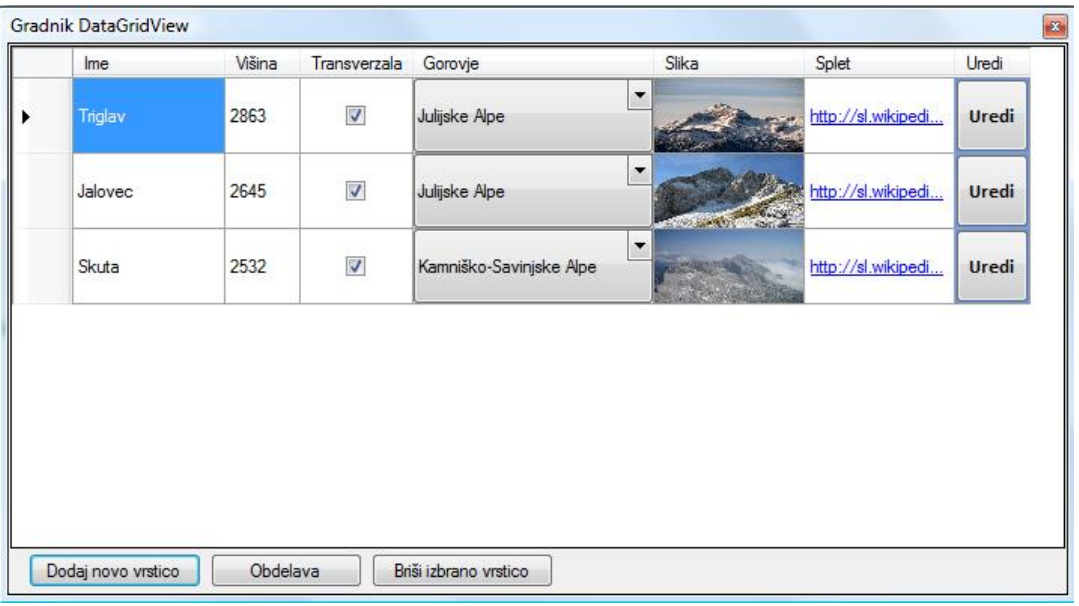
```
        Clipboard.SetText(besedilo.SelectedText); //ga kopiramo v odložišče
}
//Prilepi
private void prilepi_Click(object sender, EventArgs e)
{
    /*po lepljenju želimo postaviti kazalnik miške na konec vstavljenega
        Besedila. Pri tem si pomagamo z lastnostma SelectionStart (indeks
        začetnega znaka označenega besedila) in SelectionLength (število
znakov
        označenega besedila)*/

    int trenutni = besedilo.SelectionStart+besedilo.SelectionLength;
    besedilo.Text=besedilo.Text.Insert(besedilo.SelectionStart,
Clipboard.GetText());
    besedilo.SelectionStart = trenutni;
}
private void izberiVse_Click(object sender, EventArgs e) //Izberi vse
{
    besedilo.SelectAll();//označim celoten tekst
}
private void pisava_Click(object sender, EventArgs e) //Pisava
{
    fontDialog1.ShowDialog();//dialog za izbiro pisave
    besedilo.Font = fontDialog1.Font;//uporabimo izbrani font
}
//Barva pisave
private void barvaPisave_Click(object sender, EventArgs e)
{
    colorDialog1.ShowDialog();//dialog za barve
    besedilo.ForeColor = colorDialog1.Color;//uporabimo izbrano barvo
}
//Leva poravnava
private void levaPoravnava_Click(object sender, EventArgs e)
{
    besedilo.TextAlign = HorizontalAlignment.Left;
}
//Sredinska poravnava
private void sredinskaPoravnava_Click(object sender, EventArgs e)
{
    besedilo.TextAlign = HorizontalAlignment.Center;
}
//Desna poravnava
private void desnaPoravnava_Click(object sender, EventArgs e)
{
    besedilo.TextAlign = HorizontalAlignment.Right;
}
//Nastavitev ozadja
private void ozadje_Click(object sender, EventArgs e)
{
    colorDialog1.ShowDialog();
    besedilo.BackColor = colorDialog1.Color;
}
//prikaz oz. skrivanje statusne vrstice
```

```
private void statusnaVrsticaToolStripMenuItem_Click(object sender, EventArgs
e)
{
    if (statusnaVrsticaToolStripMenuItem.Checked)
    {
        statusStrip1.Visible = true;
        statusnaVrsticaToolStripMenuItem.Checked = false;
    }
    else
    {
        statusStrip1.Visible = false;
        statusnaVrsticaToolStripMenuItem.Checked=true;
    }
}
```

Gradnik DataGridView

Gradnik *DataGridView* je namenjen tabelaričnemu prikazu podatkov. Vsebuje stolpce in vrstice, podatke vanj pa lahko vnesemo programsko, ali pa ga preko lastnosti *DataSource* povežemo z nekim izvorom podatkov. O povezovanju z izvorom podatkov bomo zapisali več kasneje, na tem mestu pa poglejmo, kako naredimo stolpce, ter kako podatke v ta gradnik dodajamo, jih ažuriramo in brišemo programsko.

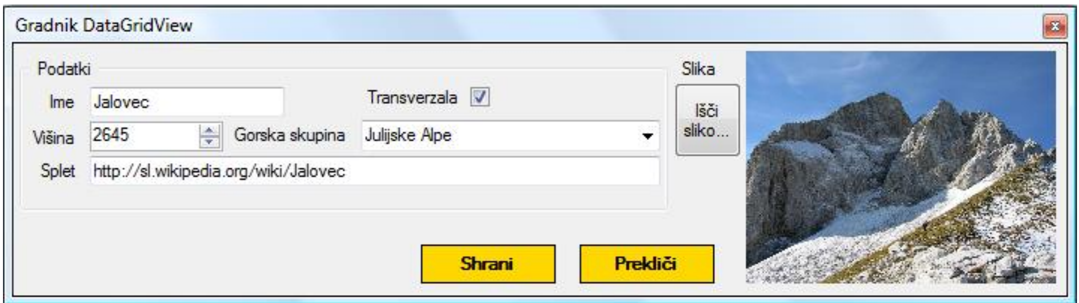
Gradnik *DataGridView*, njegove lastnosti in metode bomo najhitreje spoznali kar na konkretnem primeru. Sestavili bomo tabelarični prikaz podatkov in slik o nekaterih naših gorah.



Slika 52: Prikaz začetnega obrazca projekta *DataGridViewDemo*.

Naš program bo omogočal tudi dodajanje novih podatkov (vrstic), ter brisanje že obstoječih. Ob kliku na gumb *Obdelava* pa bomo v sporočilnem oknu izpisali skupno število vrhov, ki so v transverzali, ter kolikšna je povprečna višina vseh gora v tabeli.

Ob kliku na gumb na desni strani vsake vrstice, se bo tabela s podatki začasno skrila, prikazala pa se bo plošča z gradniki za ažuriranje podatkov izbrane vrstice, tako kot kaže slika.



Slika 53: Prikaz obrazca v primeru ažuriranja podatkov.

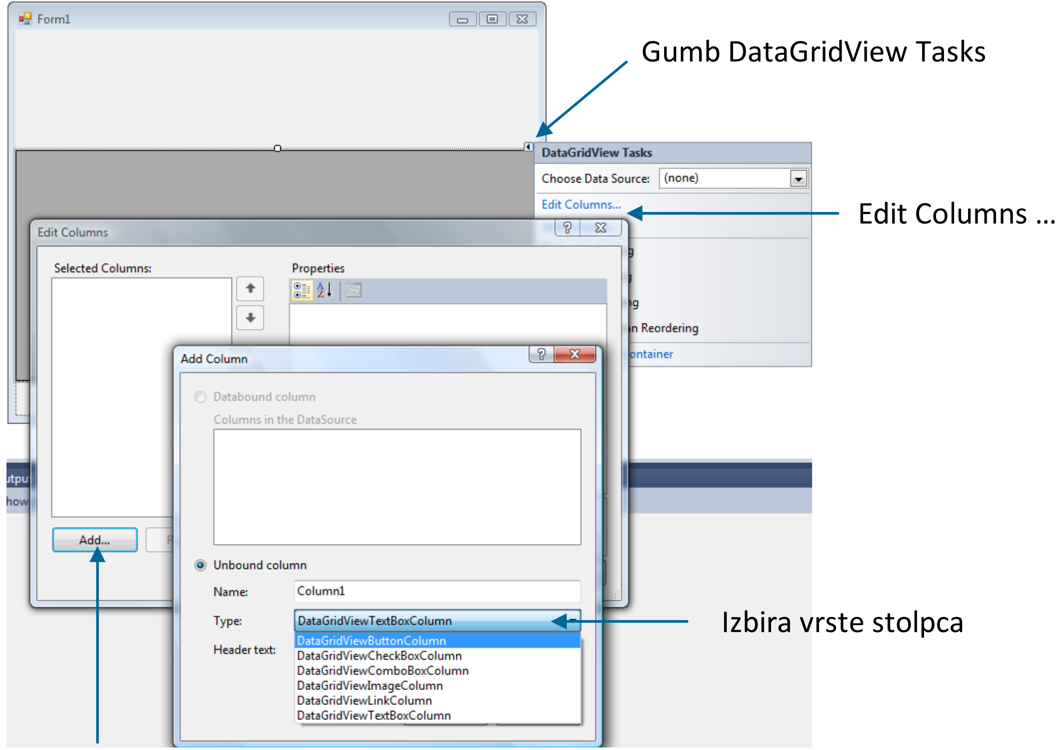
Ker v našem primeru gradnika *DataGridView* ne bomo povezovali z izvorom podatkov, bomo imena stolpcev, v katerih bomo prikazovali podatke, ustvarili že pri načrtovanju aplikacije. Na obrazec najprej postavimo gradnik *Panel* in mu nastavimo lastnost *Dock* na *Bottom*. Nanj dodajmo gumbe z napisi *Dodaj novo vrstico* in *Obdelava* in *Briši izbrano vrstico*.

Nad ploščo postavimo še gradnik *DataGridView*. Ta se nahaja na paleti *Data*. Da bo poravnan z dnom plošče, mu lastnost *Dock* nastavimo na *Bottom*. Nato pa kliknemo gumb *DataGridView Tasks* v zgornjem desnem kotu gradnika. V oknu, ki se odpre, izberemo *EditColumns*, nato pa s klikom na gumb *Add* dodajamo stolpce.

Polje *Name* predstavlja ime stolpca znotraj našega programa, lastnost *HeaderText* pa napis na vrhu stolpca. Izbrati moramo še vrsto stolpca (*Type*):

- *DataGridViewButtonColumn*: v celici bo prikazan gumb;
- *DataGridViewCheckBoxColumn*: v celici bo prikazan gradnik *CheckBox*;
- *DataGridViewComboBoxColumn*: v celici bo prikazan gradnik *ComboBox*;
- *DataGridViewImageColumn*: v celici bo prikazana slika;
- *DataGridViewLinkColumn*: v celici bo prikazana povezava (*Link*) do spletne strani ali do poljubne aplikacije;
- *DataGridViewTextBoxColumn*: celica bo namenjena za hranjenje oz vnos besedila.

Za vajo ustvarimo 7 stolpcev: *Ime* in *Visina* (*DataGridViewTextBoxColumn), Transverzala* (*DataGridViewCheckBoxColumn*), *Gorovje* (*DataGridViewComboBoxColumn*), *Slika* (*DataGridViewImageColumn*), *Splet* (*DataGridViewLinkColumn*) in *Urejanje* (*DataGridViewButtonColumn*).

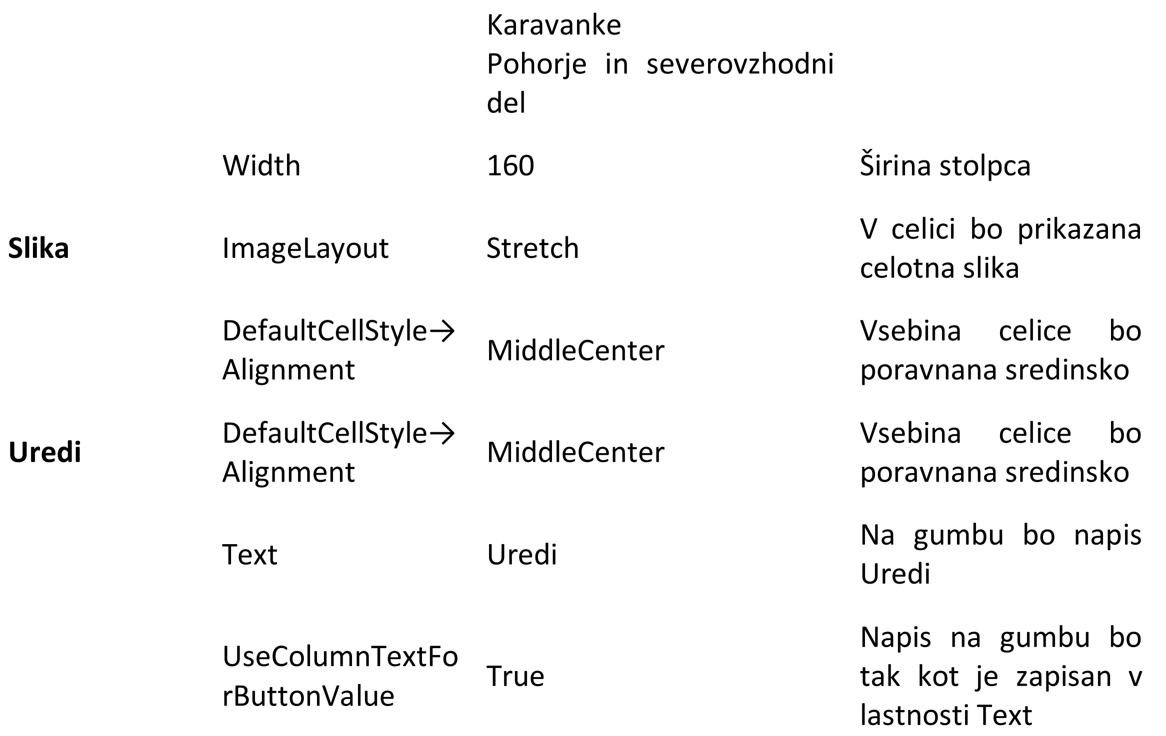


Add - Dodajanje stolpca

Slika 54: Ustvarjanje imen stolpcev gradnika *DataGridView*.

Tako ustvarjenim stolpcem določimo še nekaj osnovnih lastnosti. Odpremo urejevalnik stolpcev (ikona *DataGridView Tasks* v zgornjem desnem delu gradnika *DataGridView→Edit Columns*), nato izberemo ustrezen stolpec in določimo naslednje lastnosti:

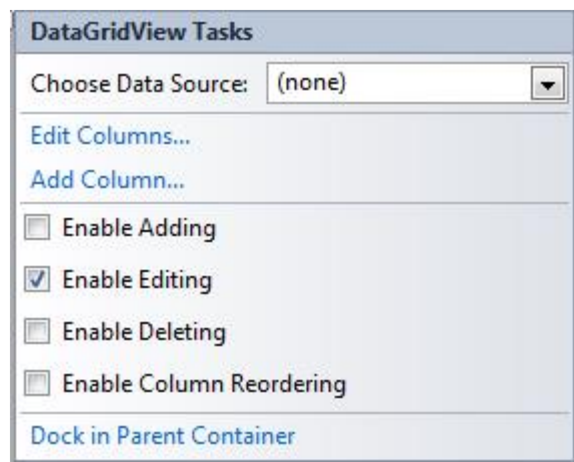
Ime stolpca	Lastnost	Nastavitev	Opis
Ime	DefaultCellStyle→ Alignment	MiddleLeft	Vsebina celice bo poravnana levo
Visina	DefaultCellStyle→ Alignment	MiddleRight	Vsebina celice bo poravnana desno
	Width	50	Širina stolpca
Transverzala	DefaultCellStyle→ Alignment	MiddleCenter	Sredinska poravnava vsebine celice
Transverzala	Width	75	Širina stolpca
Gorovje	DefaultCellStyle→ Alignment	MiddleLeft	MiddleCenter
	Items	Jugozahodni del Julijske Alpe Kamniško-Savinjske Alpe	Vsebina gradnika



		Karavanke Pohorje in severovzhodni del	
	Width	160	Širina stolpca
Slika	ImageLayout	Stretch	V celici bo prikazana celotna slika
	DefaultCellStyle→Alignment	MiddleCenter	Vsebina celice bo poravnana sredinsko
Uredi	DefaultCellStyle→Alignment	MiddleCenter	Vsebina celice bo poravnana sredinsko
	Text	Uredi	Na gumbu bo napis Uredi
	UseColumnTextForButtonValue	True	Napis na gumbu bo tak kot je zapisan v lastnosti Text

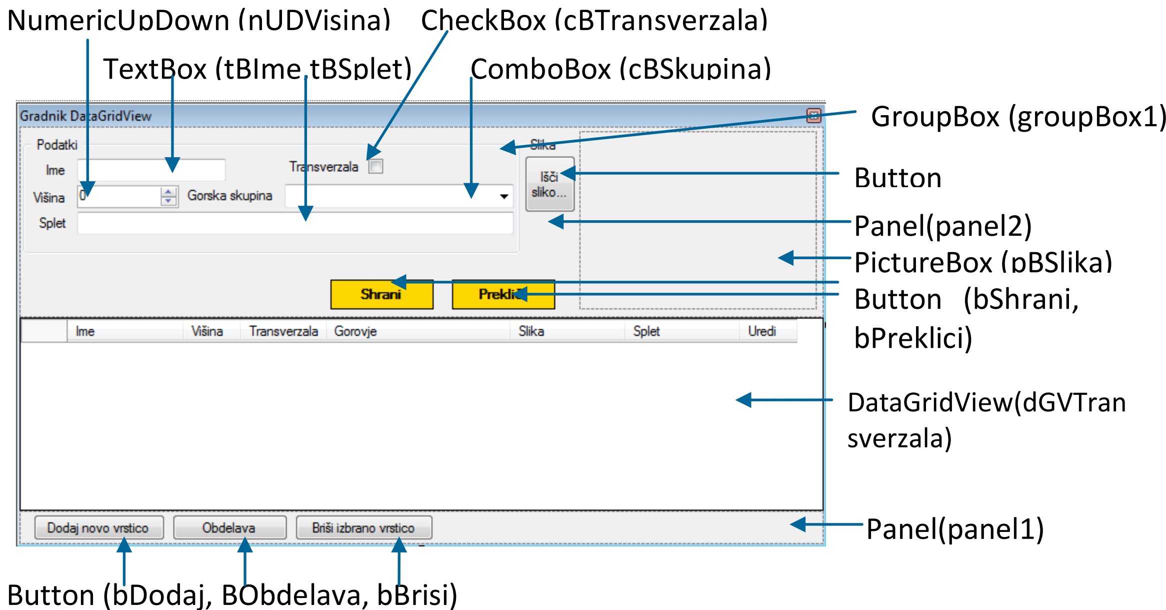
Tabela 17: Lastnosti stolpcev gradnika *DataGridView*.

Urejevalnik stolpcev nato zapremo, v oknu *DataGridVew Tasks* pa odkljukajmo le možnost *Enable Editing* (dovoljeno urejanje neposredno v posamezni celici). Ostale nastavitve pomenijo, da ni dovoljeno dodajanje vrstice neposredno v gradniku *DataGridView*, prav tako pa ni dovoljeno neposredno brisanje vrstic.



Slika 55: Nastavitev v oknu *DataGridView Tasks*.

Nad gradnik *DatagridView* postavimo gradnik *Panel* (*Dock=Fill*), nanj pa še nekaj gradnikov, ki jih že poznamo. Namenjeni bodo urejanju in dodajanju novih postavk. Projekt bomo zastavili tako, da bo uporabnik na začetku videl le postavke v gradniku *DataGridView* (zgornji panel z gradniki za urejanje bo skrit).



Slika 56: Gradniki projekta *DataGridViewDemo*.

V mapo *Bin→Debug* našega projekta skopirajmo tri slike naših vrhov, nato pa v konstruktorju obrazca poskrbimo, da bo na začetku v gradniku že nekaj vrstic podatkov. Stavki, zapisani v konstruktorju, se bodo namreč izvedli ob ustvarjanju obrazca, preden se bo ta prvič prikazal. Posamezno vrstico dodamo s pomočjo metode *Rows.Add()*, ki ima toliko parametrov, kot je stolpcev v tabeli.

```
//dodajanje nove vrstice v gradnik DataGridView
dGVTransverzala.Rows.Add("Triglav", 2863, true, "Julijske Alpe", slika,
"http://sl.wikipedia.org/wiki/Triglav");
```

Če se bo uporabnik odločil za urejanje oz. dodajanje novih zapisov, bo viden le panel z gradniki za vnos oz. ažuriranje podatkov izbrane vrstice gradnika *DataGridView*. Skrivanje in prikazovanje panela ter gradnika *DataGridView* najlaže dosežemo z metodama *BringToFront*() in *SendToBack*() teh dveh gradnikov (a le v primeru, da smo gradnikoma nastavili lastnost *Dock*). Dodajmo še osnovne metode za delo z gradnikom *DataGridView* (dodajanje, dostop do izbrane vrstice in celice, obdelava vseh vrstic, brisanje). Novo vrstico dodamo s pomočjo metode *Rows.Add()*, vsebino izbane celice tekoče vrstice pa dobimo preko objekta *CurrentRow.Cells[ime].Value* (*ime* je tukaj ime stolpca, namesto imena stolpca pa lahko zapišemo tudi njegov indeks).

Razložimo še dogodka, s katerima lahko kontroliramo uporabnikove vnose in se tem zavarujemo pred napačnimi vnosi podatkov. To sta dogodka *CellValidating* in *CellEndEdit*.

Namen metode *CellValidating* je zagotoviti, da bo uporabnik v določeno polje vnesel neko veljavno vrednost, npr. nenegativno celo število. Drugi parameter te metode, parameter *e*, je objekt tipa *DataGridViewCellValidatingEventArgs*. Ta objekt pozna tudi lastnost, s katero lahko ugotovimo, katera celica je bila ažurirana: *e.ColumnIndex* vsebuje številko stolpca, *e.RowIndex* pa številko vrstice. Pri tem upoštevamo, da imata prva vrstica in prvi stolpec indeks enak 0.

Pri obdelavi dogodka *CellValidating* lahko uporabimo metodo *int.TryParse*, ki je zelo koristna v primerih preverjanja, ali lahko nek niz pretvorimo v celo število. Metoda vrne *True*, če je pretvarjanje uspešno, obenem pa v svoj drugi parameter (v našem primeru *zacasna*), ki je klican po referenci, zapiše pretvorjeno vrednost. Če je pretvarjanje neuspešno (uporabnik je vnesel npr. niz, ki vsebuje črke, ali pa je vnesena vrednost negativna), se v lastnost *ErrorText* trenutne vrstice v tabeli zapiše ustrezno obvestilo o napaki. Na začetku te vrstice se prikaže ikona s klicajem. Če na ikono postavimo kazalnik miške, se pod njo izpiše obvestilo o napaki (v našem primeru tekst "*Vrednost mora biti nenegativno število*"). Z naslednjim stavkom (*e.Cancel = true;*) pa poskrbimo, da se uporabnik nikakor ne more premakniti v drugo celico vse dotlej, dokler v celico ne vnese pravilnega podatka, ali pa celoten vnos prekliče s tipko *Esc*.

Za gradnik *DataGridView* zapišimo še odzivno metodo za dogodek *CellEndEdit*. Ta metoda se izvede, ko je uporabnikov vnos veljaven in se uporabnik premakne v drugo celico. Koda je naslednja:

```
dGVTransverzala.Rows[e.RowIndex].ErrorText = "";
```

S tem stavkom enostavno pobrišemo vsa obvestila o napaki zaradi nepravilnega vnosa podatkov v trenutno izbrano celico.

Pojasnimo še namen dogodka *DataError* gradnika *DataGridView*. Ta dogodek zajame prav vse napake, ki nastanejo pri preverjanju uporabnikovega vnosa v katerokoli celico (predvsem velja to za celice, katerih vneseno vsebino nismo preverjali preko imena celice. Vsebina ustrezne metode poskrbi za splošno obvestilo uporabniku, obenem pa prepreči, da bi se uporabnik kljub napačnemu vnosu premaknil iz trenutne celice.

```
public partial class Form1 : Form
{
    /*spremenljivka urejanje določa, ali je gradnik DataGridView v fazi
      urejanja (true) ali dodajanja (false)*/
    bool urejanje;
    public Form1() //konstruktor obrazca Form1
    {
        InitializeComponent();
        //na začetku je panel skrit
        panel2.Hide();
        //DataGridView razširimo čez celoten obrazec
        dGVTransverzala.Dock = DockStyle.Fill;

        //določimo višino posameznih vrstic (zaradi slik)
        dGVTransverzala.RowTemplate.Height = 50;
        //v DataGridView z metodo Items.Add dodamo tri vrstice
        Image slika = Image.FromFile("Triglav.JPG");
        dGVTransverzala.Rows.Add("Triglav", 2863, true, "Julijske Alpe",
slika, "http://sl.wikipedia.org/wiki/Triglav");
        slika = Image.FromFile("Jalovec.JPG");
        dGVTransverzala.Rows.Add("Jalovec", 2645, true, "Julijske Alpe",
slika, "http://sl.wikipedia.org/wiki/Jalovec");
        slika = Image.FromFile("Skuta.JPG");
        dGVTransverzala.Rows.Add("Skuta", 2532, true, "Kamniško-Savinjske
Alpe", slika, "http://sl.wikipedia.org/wiki/Skuta");
```

```
    }
    /*ob kliku na celico v gradniku DataGridView se najprej zgodi dogodek
      CellClick, za njim pa še dogodek CellContentClick. Samo za informacijo:
      dogodek CellClick se zgodi pri vsakem kliku v notranjost celice,
      CellContentClick pa le, če kliknemo na neko vsebino v celici */
    private void dGVTransverzala_CellClick(object sender,
DataGridViewCellEventArgs e)
    {
        urejanje = true;
        if (e.ColumnIndex == this.dGVTransverzala.Columns["Uredi"].Index)
        {
            //vsebino posameznih celic zapišemo v ustrezne gradnike
            tBime.Text =
dGVTransverzala.CurrentRow.Cells["Ime"].Value.ToString();
            nUDVisina.Value =
Convert.ToInt32(dGVTransverzala.CurrentRow.Cells["Visina"].Value);
            cBTransverzala.Checked =
Convert.ToBoolean(dGVTransverzala.CurrentRow.Cells["Transverzala"].Value);
            Image slika = (Image)
(dGVTransverzala.CurrentRow.Cells["Slika"]).Value;
            pBSlika.Image = slika;
            tBSplet.Text =
dGVTransverzala.CurrentRow.Cells["Splet"].Value.ToString();
            cBSkupina.Text =
dGVTransverzala.CurrentRow.Cells["Gorovje"].Value.ToString();
            //prikažemo panel in s tem skrijemo DataGridView
            panel2.Show();
            panel1.Visible = false;
            this.Height = 190;
        }
    }
    private void bShrani_Click(object sender, EventArgs e)
    {
        if (urejanje)
        {
            try
            {
                /*vrednosti gradnikov na panelu Panel2 prenesemo v
                  vrstico gradnika DataGridView*/
                dGVTransverzala.CurrentRow.Cells["Ime"].Value = tBime.Text;
                dGVTransverzala.CurrentRow.Cells["Visina"].Value =
                   nUDVisina.Value;
                dGVTransverzala.CurrentRow.Cells["Transverzala"].Value =
                   cBTransverzala.Checked;
                dGVTransverzala.CurrentRow.Cells["Splet"].Value =
                   tBSplet.Text;
                dGVTransverzala.CurrentRow.Cells["Gorovje"].Value =
                   cBSkupina.Text;
                dGVTransverzala.CurrentRow.Cells["Slika"].Value =
                  pBSlika.Image;
            }
            catch
            {  MessageBox.Show("Napaka pri shranjevanju!"); }
```

```
            finally
            {
                /*po urejanju skrijemo Panel2*/
                panel2.Hide();
                //prikaz panela z gumboma za urejanje in dodajanje
                panel1.Visible = true;
                urejanje = false;//oznaka, da je urejanje končano
            }
        }
        else
        {
            /*najprej preverimo, če je uporabnik vnesel vse podatke
            nato pa jih shranimo v gradnik DataGridView*/
            dGVTransverzala.Rows.Add(tBime.Text, nUDVisina.Value,
cBTransverzala.CheckState, cBSkupina.Text, pBSlika.Image, tBSplet.Text);
            panel1.Visible = true;
            urejanje = false;
            /*skrijemo Panel2*/
            panel2.Hide();
        }
        this.Height = 400;
    }

    private void bPreklici_Click(object sender, EventArgs e)
    {
        /*po urejanju skrijemo Panel2*/
        panel2.Hide();
        //prikažemo panel z gumboma za urejanje in dodajanje
        panel1.Visible = true;
        urejanje = false;
        this.Height = 400;
    }

    private void bIsciSliko_Click(object sender, EventArgs e)
    {
        try
        {
            //ustvarimo nov objekt tipa OpenFileDialog
            OpenFileDialog OPF = new OpenFileDialog();
            //v oknu bodo prikazane le slikovne datoteke
            OPF.Filter = "Slike|*.jpg;*.gif;*.bmp;*.pmg";
            //okno zapremo s sklikom na gumb Open
            if (OPF.ShowDialog() == DialogResult.OK)
            {
                //izbrano sliko prenesemo v gradnik
                Image slika = Image.FromFile(OPF.FileName);
                pBSlika.Image = slika;
            }
        }
        catch
        { MessageBox.Show("Napaka!"); }
    }
```

```
    private void dGVTransverzala_CellContentClick(object sender,
DataGridViewCellEventArgs e)
    {
        //preverim, če je bila kliknjena prava celica
        if (e.ColumnIndex == this.dGVTransverzala.Columns["Splet"].Index)
            System.Diagnostics.Process.Start("IExplore",
dGVTransverzala.CurrentRow.Cells["Splet"].Value.ToString());
    }
    private void bDodaj_Click(object sender, EventArgs e)
    {
        panel2.Show();
        panel1.Visible = false;
        //pobrišemo dosedanje vrednosti gradnikov
        tBime.Clear(); nUDVisina.Value = 0; tBSplet.Clear();
        cBSkupina.Text = "";
        cBTransverzala.Checked = false;
        pBSlika.Image = null;
        urejanje = false;
        this.Height = 190;
    }

    //odzivna metoda gumba za obdelavo tabele
    private void button1_Click(object sender, EventArgs e)
    {
        //ugotovimo, koliko je skupno število vrhov, ki so v transverzali
        int skupaj = 0;
        //ugotovimo še, kolikšna je povprečna višina vseh gora
        int visina = 0;
        for (int i = 0; i < dGVTransverzala.Rows.Count; i++)
        {
            if
(Convert.ToBoolean(dGVTransverzala.Rows[i].Cells["Transverzala"].Value))
                skupaj++;
            //prištevamo vrednost v celici Visina

visina=visina+Convert.ToInt32(dGVTransverzala.Rows[i].Cells["Visina"].Value);
        }
        MessageBox.Show("Skupaj gora v transverzali: "+skupaj+"\nPovprečna
višina vseh gora: "+Math.Round((double)visina/dGVTransverzala.Rows.Count,2)+"
m");
    }

    private void dGVTransverzala_CellValidating(object sender,
DataGridViewCellValidatingEventArgs e)
    {
        int zacasna;
        dGVTransverzala.Rows[e.RowIndex].ErrorText = "";
        /*če smo naredili spremembe v stolpcu Visina, je potrebna
          validacija*/
        if ((dGVTransverzala.Columns[e.ColumnIndex].Name == "Visina"))
        {
            /*metoda TryParse pretvarja niz v celo število. Ima dva
parametra: prvi je tipa niz, drugi pa celo število(klic po referenci). Če
```

```
pretvorba uspe, metoda vrne true, rezultat pretvorbe pa je shranjen v drugem
parametru*/
            if (!int.TryParse(e.FormattedValue.ToString(), out zacasna) ||
zacasna < 0)
            {
                /*v primeru napake oblikujemo ustrezno sporočilo: potrebno
                  pa je napisati dogodek CellEndEdit*/
                dGVTransverzala.Rows[e.RowIndex].ErrorText = "Vrednost mora
biti nenegativno število";
                /*uporabnik se nikakor ne more premakniti v drugo celico
                  vse dotlej, dokler v celico ne vnese veljavnega podatka*/
                e.Cancel = true;
            }
        }
    }

    private void dGVTransverzala_CellEndEdit(object sender,
DataGridViewCellEventArgs e)
    {
        //ko je napaka odpravljena izbrišemo sporočilo o napaki
        dGVTransverzala.Rows[e.RowIndex].ErrorText = "";
    }
    //odziva metoda gumba za brisanje vrstice gradnika ataGridView
    private void button2_Click(object sender, EventArgs e)
    {
        if (MessageBox.Show("Brišem izbrano
vrstico?","Brisanje?",MessageBoxButtons.OKCancel,MessageBoxIcon.Question)==Di
alogResult.OK)
        {
            DataGridViewRow row = dGVTransverzala.CurrentRow;
            dGVTransverzala.Rows.Remove(row);
        }
    }
}
```

Pokažimo še, kako lahko gradnik *DataGridView* povežemo z nekim izvorom podatkov. O povezavi z neko tabelo iz baze bomo več napisali pri bazah podatkov, na tem mestu pa bomo pokazali povezavo z neko tipizirano zbirko.

V matematiki pogosto delamo s pravokotnimi trikotniki. Sestavimo razred *PravokotniTrikotnik* z dvema poljema (kateti pravokotnega trikotnika), metodo za izračun ploščine trikotnika, ter lastnostmi za izračun hipotenuze in za nastavljanje vrednosti katet.

```
public class PravokotniTrikotnik
{
    double a, b;  //kateti sta zasebni polji
    public PravokotniTrikotnik(double a, double b) //konstruktor
    {
        this.Kateta_a = a;
        this.Kateta_b = b;
    }
```

```
    //Lastnosti
    public double Kateta_a //lastnost za prvo kateto
    {
        get { return a;}
        set
        {   //če bomo za a skušali nastaviti negativno vrednost, ostane a=0
            if (value > 0) a = value;
        }
    }
    public double Kateta_b //lastnost za drugo kateto
    {
        get { return b; }
        set
        {   //če bomo za b skušali nastaviti negativno vrednost, ostane b=0
            if (value >= 0) b = value;
        }
    }

    public double Hipotenuza //lastnost za hipotenuzo
    {
        get { return Math.Round(Math.Sqrt(a * a + b * b), 2);}
    }
    public double Ploščina //lastnost za ploščino pravokotnega trikotnika
    {
        get { return a * b / 2.0;}
    }
}
```

V konstruktorju obrazca ustvarimo 100 objektov tipa *PravokotniTrikotnik* (kateti naj bosta naključni celi števili med 1 in 100) in objekte dodajmo v tipizirano zbirko *trikotniki*. Zbirko nato povežimo z gradnikom *DataGridView* (ime gradnika je *DGV*), ki ga pred tem postavimo na prazen obrazec.

```
//Deklaracija tipizirane zbirke pravokotnih trikotnikov
List<PravokotniTrikotnik> trikotniki = new List<PravokotniTrikotnik>();
//generator naključnih števil
Random naklj = new Random();

//Konstruktor obrazca
public Form1()
{
    InitializeComponent();
    /*v zanki ustvarimo 100 objektov tipa PravokotniTrikotnik in jih dodamo
     v zbirko*/
    for (int i = 0; i < 100; i++)
    {
        //kateti naj bosta naključni celi števili med 1 in 100
        int katetaA=naklj.Next(1,101);
        int katetaB=naklj.Next(1,101);
        PravokotniTrikotnik p = new PravokotniTrikotnik(katetaA, katetaB);
        //objekt p dodamo v zbirko trikotniki
        trikotniki.Add(p);
    }
```

```
    /*objekt DataGridView (ime gradnika je DGV in je že na obrazcu),
      povežemo z tipizirano zbirko Trikotniki*/
    DGV.DataSource = trikotniki;
    //nekatere oblikovne lastnosti gradnika DGV lahko nastavimo programsko
    DGV.Columns[0].DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleCenter;
    DGV.Columns[1].DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleCenter;
    DGV.Columns[2].DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleCenter;
    DGV.Columns[3].DefaultCellStyle.Alignment =
DataGridViewContentAlignment.MiddleRight;

    //oblikujmo font za glave stolpcev
    DataGridViewCellStyle columnHeaderStyle = new DataGridViewCellStyle();
    columnHeaderStyle.ForeColor = Color.Red;
    columnHeaderStyle.Font = new Font("Verdana", 8, FontStyle.Bold);
    DGV.ColumnHeadersDefaultCellStyle = columnHeaderStyle;
    //določimo še širino obrazca
    this.Width = DGV.Columns[0].Width * 5;
    DGV.ReadOnly = true;//urejanje izključimo zaradi povezave s podat.izvorom
}
```

Stolpci gradnika *DGV* se pri povezavi ustvarijo avtomatično, njihova imena pa so enaka lastnostim razreda *PravokotniTrikotnik*. V zgornjem primeru so vsi stolpci *ReadOnly*, kar pomeni, da uporabnik vrednosti celic neposredno ne more spreminjati. S tem se izognemo napakam pri spreminjanju podatkov o obeh katetah (stolpca *Hipotenuza* in *Ploščina* pa tako in tako ne moremo spreminjati, ker predstavljata lastnosti).

Pravokotni trikotnik

Kateta_a	Kateta_b	Hipotenuza	Ploščina
1	62	62,01	31
2	10	10,2	10
16	8	17,89	64
88	62	107,65	2728
11	78	78,77	429
67	44	80,16	1474
89	58	106,23	2581
46	93	103,75	2139
86	28	90,44	1204
71	97	120,21	3443,5

Slika 57: Prikaz vsebine tipizirane zbirke v gradniku *DataGridView*.

Namesto zbirke tipa *List* lahko uporabimo tudi zbirko tipa *BindingList*. Prednost zbirke tipa *BindingList* je v tem, da se vsaka sprememba v zbirki takoj odrazi tudi v gradniku *DataGridView* in obratno. Delo z obema zbirkama je sicer povsem enako.

Dogodki tipkovnice, miške in ure

Dogodki tipkovnice

Obrazci in večina gradnikov v okolju Windows obravnavajo vnose preko tipkovnice z obdelavo dogodkov ki jih proži tipkovnica. V oknu *Properties→Events* teh gradnikov obstajata dva dogodka, ki se zgodita, ko uporabnik pritisne tipko na tipkovnici in en dogodek ko spusti tipko na tipkovnici. Dogodki tipkovnice so predstavljeni in razloženi v naslednji tabeli:

Dogodek Tipkovnice	Razlaga
KeyDown	Dogodek, ki se izvede ob pritisku na katerokoli tipko. Zgodi se samo enkrat.
KeyPress	Dogodek se zgodi ob pritisku na tipko, ki predstavlja nek znak iz množice vseh znakov. Če uporabnik tipko drži pritisnjeno, se zgodi večkrat.
KeyUp	Dogodek se zgodi, ko uporabnik spusti tipko.

Tabela 18: Dogodki tipkovnice.

Ko uporabnik pritisne tipko, se najprej preverja, kateri dogodek se bo izvedel. To pa je odvisno od tega, ali je pritisnjena tipka (ali pa kombinacija tipk) nek ASCII znak (črke, števke, *Enter*, *BkSp, Shift A*, *AltGr W*…), ali pa gre za kako drugo tipko na tipkovnici (npr. *Tab*, *CapsLock*, *Shift, Ctrl, Insert, Delete, PgUp, PgDn* …). Če kombinacija tipk, ki jih uporabnik pritisne, ustreza nekemu znaku, se najprej zgodi dogodek *KeyPress*, za njim pa še dogodek *KeyDown*. Če ne gre za znakovno tipko (npr. tipka *Shift*), se zgodi le dogodek *KeyDown*.

Vrstni red dogodkov, ki jih proži tipkovnica je torej naslednji:

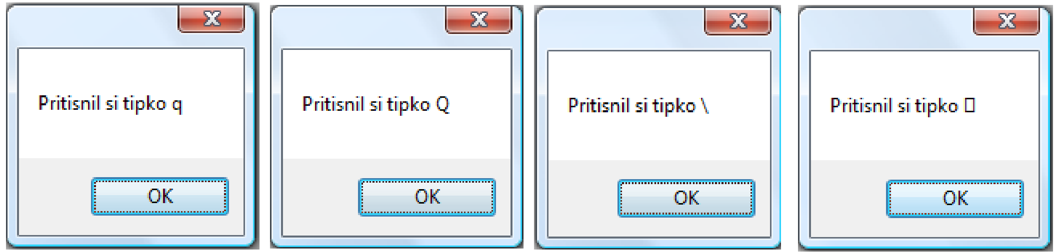
- *KeyPress* – izvede se ob pritisku na tipko, ki predstavlja nek znak iz množice vseh znakov. Dogodek se bo sprožil če npr. uporabnik pritisne tipko *'Q'*, če uporabnik pritisne kombinacijo tipk *Shift + 'Q'*, če npr. uporabnik pritisne kombinacijo tipk *AltGr + 'Q'*, če pritisne tipko *Ctrl+'Q'* … Vsaka od napisanih kombinacij namreč predstavlja nek znak, ki ga dobimo s pomočjo tipkovnice.
- *KeyDown* – izvede se ob pritisku na katerokoli tipko.
- *KeyUp* – izvede se, ko uporabnik spusti katerokoli tipko.

Za vsako tipko na tipkovnici lahko preko drugega parametra odzivnega dogodka tipkovnice dobimo vse podatke o tej tipki ali pa kombinaciji tipk:

- Dogodek *KeyPress* ima drug parameter tipa *KeyPressEventArgs*. Objekt tega razreda nam pove, katero tipko ali pa katero kombinacijo tipk je pritisnil uporabnik. Oznako tipke, ali pa kombinacijo tipk lahko izpišemo v sporočilnem oknu s pomočjo lastnosti KeyChar: ta predstavlja znak, ki ustreza pritisnejeni tipki.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{     MessageBox.Show("Pritisnil si tipko " + e.KeyChar);}
```

Spodnja slika prikazujejo sporočilna okna, ko smo zaporedoma pristisnili tipko *'Q'*, kombinacijo tipk *Shift + 'Q'*, kombinacijo tipk *AltGr + 'Q'* in kombinacijo *Ctrl+'Q'*.



Slika 58: Sporočilna okna s prikazom različnih kombinacij pritiska tipke Q.

- Dogodek *KeyDown* ima drug parameter tipa *KeyEventArgs*. To je razred, ki vsebuje nekaj zelo koristnih metod, s katerimi lahko kontroliramo uporabnikov pritisk na katerokoli tipko na tipkovnici. Hrani tudi podatka o imenu tipke in o kodi ASCII tega znaka. Podatka dobimo s pomočjo lastnosti *KeyCode* in *KeyValue*. Njuni vrednosti lahko npr. izpišemo v sporočilnem oknu:

```
Private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
    MessageBox.Show("Pritisnil si tipko "+ e.KeyCode + "- ASCII: = "
    + e.KeyValue);
}
```

S pomočjo drugega parametra in lastnosti *Handled*, lahko dogodek predčasno zaključimo in na ta način kontroliramo uporabnikove vnose preko tipkovnice. S tem stavkom enostavno povemo, da so se vsi dogodki, povezani s to tipko, že zaključili, zato se pritisnjena tipka v gradniku sploh ne prikaže.

```
e.Handled = true; /dogodek je zaključen, vsi nadaljnji stavki v telesu
dogodka se ne bodo izvedli*/
```

- Dogodek *KeyUp* ima drug parameter tipa *KeyEventArgs*. Z njegovo pomočjo dobimo podatek, katera tipka je bila spuščena. Z lastnostjo *Handled* lahko tudi v tem dogodku le-tega predčasno zaključimo, ali ga celo preprečimo.

```
private void tbStarost_KeyUp(object sender, KeyEventArgs e)
{
    MessageBox.Show("Spustil si tipko " + e.KeyCode);
}
```

Opisane dogodke pogostokrat uporabljamo za kontrolo uporabnikovih vnosov podatkov. Tako lahko npr. uporabniku preprečimo vnos znakov, ki niso številke, v tisto vnosno polje, ki naj bi predstavljalo nek znesek, ali pa vnos numeričnih podatkov v vnosno polje, ki naj bi predstavljalo zaporedje črk abecede, ipd.

Kadar torej želimo specifične vnose, ali pa uporabniku dovoliti uporabo le določenih tipk, oz. mu preprečiti nezaželene vnose, lahko to storimo s pomočjo dogodka *KeyPress*! Tule je nekaj primerov odzivnih dogodkov vrste *KeyPress*, ko uporabnik vnaša podatke v gradnik tipa *TextBox*.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    //Uporabniku dovolimo le vnos znakov angleške abecede:
    if ((e.KeyChar < 'A') || (e.KeyChar > 'Z')) e.Handled = true;
}
```

Zopet smo uporabili drugi parameter odzivne metode in lastnost Handled: vsi dogodki, povezani s to tipko so zaključeni. Celotni zgornji stavek moramo torej razumeti takole: če pritisnjena tipka ni velika črka med A in Z, potem velja, kot da nismo pritisnili nobene tipke.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    //Uporabniku dovolimo le vnos znakov D ali N:
    if ((e.KeyChar != 'D') && (e.KeyChar != 'N')) e.Handled = true;
}
```

Če torej nismo pritisnili velike črke D oziroma velike črke N, potem velja, kot da nismo pritisnili nobene tipke.

Pa še primer kode odzivnega dogodka, s katerim uporabniku dovolimo le vnos števk in decimalne vejice, dovolimo pa mu tudi brisanje že napisanih znakov (torej uporabo tipke *BackSpace*).

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    /*Uporabniku dovolimo le vnos števk in decimalne vejice, dovolimo pa mu
      tudi brisanje že napisanih znakov ( (char)8 je tipa BackSpace)*/
    if (((e.KeyChar < '0') || (e.KeyChar > '9'))
        && (e.KeyChar != ',') && (e.KeyChar != (char)(8))) e.Handled = true;
}
```

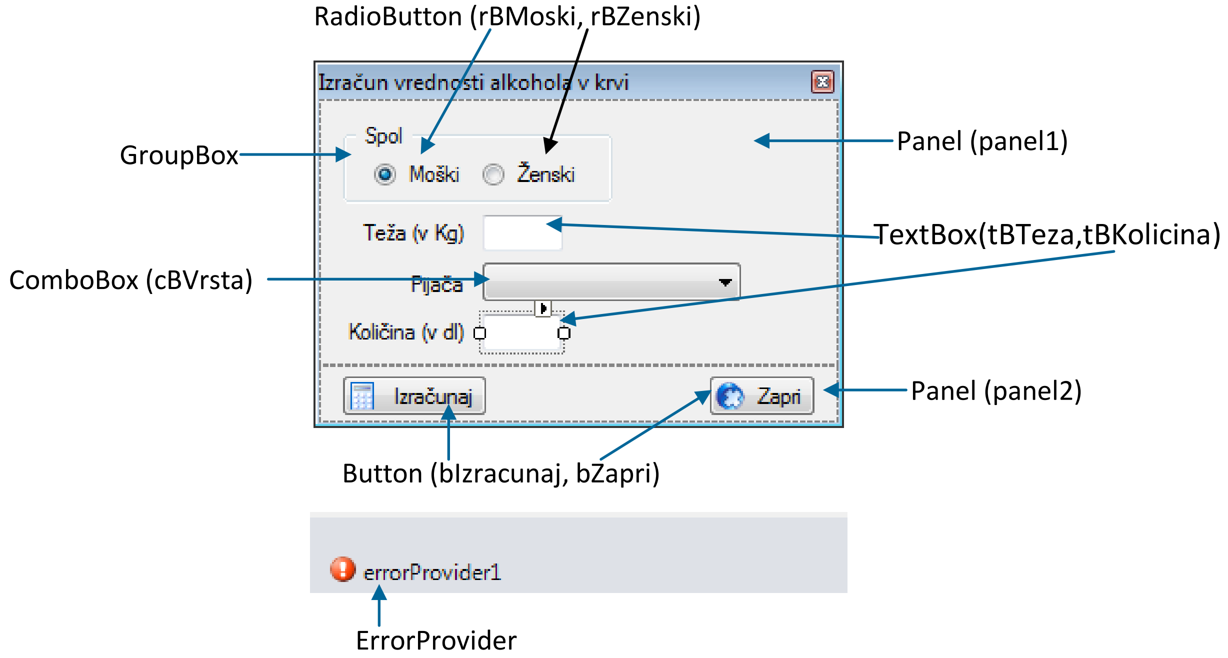
Omenjene dogodke bomo uporabili pri programiranju naslednjega zgleda.

Izračun stopnje alkohola v krvi

Približen čas izločanja alkohola iz telesa je povprečno od 0,1 do 0,15 g za vsak kg teže osebe na uro. Če imamo podatek o koncentraciji alkohola v krvi, lahko izračunamo koncentracijo alkohola v krvi. Poenostavljena formula za približen izračun koncentracije je: *c = m / (TT)*r*, pri čemer je:

c = koncentracija alkohola v krvi (gramov oz promilov etanola na kg krvi)
m = masa popitega čistega alkohola izražena v gramih
TT = telesna masa izražena v kilogramih
r = porazdelitveni faktor (za moške 0,7, za ženske 0,6)

Podatke o stopnji alkohola v posameznih vrstah pijače imamo zbrane v tekstovni datoteki *Alkohol.txt*. Napišimo program, ki bo glede na spol, vrsto pijače, količino zaužite pijače in teže izračunal, kolikšna je približna koncentracija alkohola v krvi potem, ko popijemo to vrsto pijače (in smo bili prej povsem trezni). Pri tem naj bo uporabniški vmesnik tak, kot ga prikazuje slika. Projekt poimenujmo *Alkohol*, obrazec pa *FAlkohol*. Nanj postavimo naslednje gradnike:



Slika 59: Gradniki na obrazcu *FAlkohol*.

Gradnik	Lastnost	Nastavitev	Opis
FAlkohol	FormBorderStyle	FixedToolWindow	Uporabnik velikosti okna ne more spreminjati.
panel2	Dock	Bottom	Spodnji panel je prilepljen na dno obrazca *FAlkohol*.
panel1	Dock	Fill	Zgornji panel je raztegnjen čez vso preostalo površino obrazca *FAlkohol*.

rBMoski	Checked	True	Radijski gumb je izbran.
tBTeza	TextAlign	Right	Teža bo desno poravnana.
tBKolicina	TextAlign	Right	Količina bo desno poravnana.
cBVrsta	DropDownStyle	DropDownList	Uporabnik ne bo mogel vpisati svoje vrednosti, ampak le izbral eno od obstoječih.
bIzracunaj	Image	Poljubna slika	
bIzracunaj	ImageAlign	MiddleLeft	Slika bo na gumbu poravnana na levi strani.
bIzracunaj	TextAlign	MiddleRight	Besedilo bo na gumbu poravnano na desni strani.
bZapri	Image	Poljubna slika	
bZapri	ImageAlign	MiddleLeft	Slika bo na gumbu poravnana na levi strani.
bZapri	TextAlign	MiddleRight	Besedilo bo na gumbu poravnano na desni strani.

Tabela 19: Lastnosti gradnikov obrazca *FAlkohol*.

V projektu smo ustvarili razred *pijaca* z dvema poljema (vrsta pijače in stopnja alkohola). Dogodek *Load* obrazca smo uporabili za prenos podatkov iz tekstovne datoteke *Alkohol.txt* (ta se nahaja v mapi *Bin→Debug* našega projekta) v tipizirano zbirko objektov tipa *pijaca* (vrsta zbirke je *List*) z imenom *Pijace*. Potem še gradnik *ComboBox* (imenovan *cBVrsta*) s pomočjo zbirke *Pijace* napolnimo z vrstami pijač (opisali smo ga že v programu *Anketa*).

Ker je najbolj tipičen podatek, ki ga uporabnik vnese ob zagonu programa, njegova teža, poskrbimo, da bo vnosno mesto v tem tekstovnem polju. Za to bomo poskrbeli tako, da bomo v v odzivni metodi obrazca na dogodek *Shown* poskrbeli, da bomo fokus postavili na to vnosno polje. Dogodek *Shown* se zgodi takoj za dogodkom *Load*, ko so objekti na obrazcu že ustvarjeni.

```
public FAlkohol()
{
    InitializeComponent();
}
/*razred Pijaca: vsak objekt tega tipa bo hranil podatke o vrsti pijače
  in stopnji alkohola v njej*/
public class Pijaca
{
    public string vrsta;
    public double stopnja;
    //konstruktor
    public Pijaca(string vrsta, double stopnja)
```

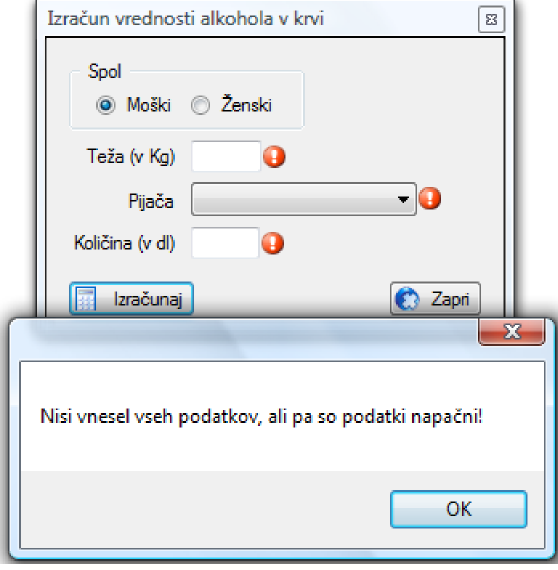
```
    {
        this.vrsta = vrsta;
        this.stopnja = stopnja;
    }
    public string Vrsta //lastnost/property
    {
        get { return vrsta; }
        set { vrsta = value; }
    }
    public double Stopnja //lastnost/property
    {
        get { return stopnja; }
        set { stopnja = value; }
    }
}
List<Pijaca> pijace = new List<Pijaca>();//deklaracija tipizirane zbirke
/*preden se obrazec prikaže na ekranu, napolnimo tako zbirko kot ComboBox
   s podatki iz datoteke*/

private void FAlkohol_Load(object sender, EventArgs e)
{
    try
    {
        //podatke iz tekstovne datoteke prenesemo  tabelo objektov
        StreamReader beri = File.OpenText("Alkohol.txt");
        string vrstica = beri.ReadLine();//beremo vrstico
        while (vrstica != null)  //če branje uspešno
        {
            /*podatka o vrsti pijače in stopnji alkohola sta v vrstici
              razmejena z znakom '|'. Ločimo ju z metodo Split*/
            string[] zac = vrstica.Split('|');
            //ustvarimo nov objekt tipa Pijaca
            pijaca nova=new pijaca(zac[0],Convert.ToDouble(zac[1]));
            pijace.Add(nova); //in ga damo  zbirko
            vrstica = beri.ReadLine();//beremo vrstico
        }
        beri.Close();//zapremo podatkovni tok/datoteko
        //gradnik cBVrsta povežemo s podatkovnim izvorom – zbirko pijace
        cBVrsta.DataSource = pijace;
        //določimo polje, ki se bo prikazovalo v spustnem seznamu
        cBVrsta.DisplayMember ="Vrsta";//Vrsta=lastnost razreda Pijaca
        //Določimo polje, ki ga bomo uporabili kot vrednost
        cBVrsta.ValueMember ="Stopnja";//Stopnja=lastnost razreda Pijaca
    }
    catch
    {   MessageBox.Show("Napaka pri delu z datoteko!"); }
}

/*dogodek Shown se zgodi takoj za dogodkom Load, ko so objekti na obrazcu
  že ustvarjeni*/
private void FAlkohol_Shown(object sender, EventArgs e)
{
```

```
    tBTeza.Focus();//Na začetku bo aktiven gradnik tBTeza
}
private void bZapri_Click(object sender, EventArgs e)
{
    Application.Exit();//zaključek projekta, obrazec se zapre
}
```

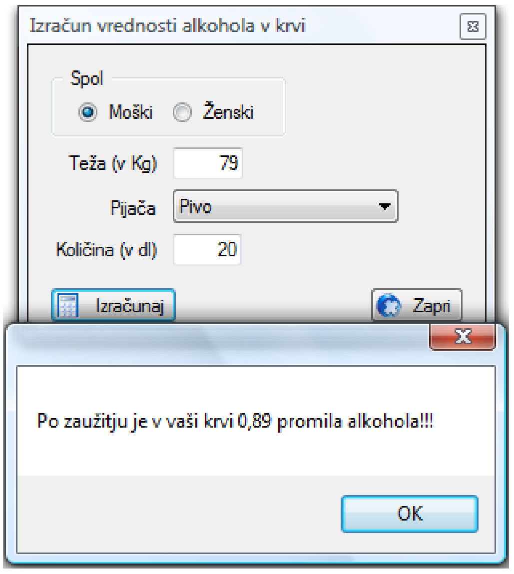
Uporabnikov vnos teže in količine kontroliramo s pomočjo dogodka *KeyPress* (dovolimo le vnos števk, decimalne vejice, dovolimo pa tudi brisanje že napisanih znakov). S pomočjo gradnika *ErrorProvider* pa še preverjamo, ali je uporabnik sploh kaj vnesel oz. ali je izbral vrsto pijače. Celotno kodo smo zapakirali še v varovalni blok in oblikovali osnovno sporočilo o napaki! Zaradi enostavnosti v razredu ni kontrole pravilnosti stanj (vrednosti polj vrsta in stopnja), saj predpostavljamo, da njihovo pravilnost program oz. kontrolira uporabniški vmesnik.

```
private void tB_KeyPress(object sender, KeyPressEventArgs e)
{
    /*če vneseni znak ni števka (0 do 9), ali decimalna vejica, ali pa
      tipka BackSpace (char 8), je dogodek zaključen*/
    if (((e.KeyChar < '0') || (e.KeyChar > '9')) && (e.KeyChar != ',') &&
(e.KeyChar != (char)(8)))
        e.Handled = true;
}

private void bIzracunaj_Click(object sender, EventArgs e)
{
    /*kontrola vnosa teže se izvede avtomatično, ker je gradnik tBTeza ob
      zagonu aktiven gradnik (ima fokus)*/
    tB_Validating(cBVrsta, null);//kontrola izbire vrste pijače
    tB_Validating(tBKolicina, null);//kontrola vnosa količini
    try
    {
        /*izračun stopnje alkohola v krvi: za izbrani tekst v gradniku
ComboBox poiščemo stopnjo alhohola v zbirki pijace*/
        int indeks = cBVrsta.SelectedIndex;
        double stopnja = Convert.ToDouble(cBVrsta.SelectedValue);
        double teza = Convert.ToDouble(tBTeza.Text);
        double skupno_gramov=(Convert.ToDouble(tBKolicina.Text))*stopnja;
        double alkohola;
        if (rBMoski.Checked)  //Izbran radijski gumb Moški
            alkohola = skupno_gramov / (teza) * 0.6;
        else alkohola = skupno_gramov / (teza) * 0.7;/*Ženske*/
        MessageBox.Show("Po zaužitju je v vaši krvi " +
Convert.ToString(Math.Round(alkohola, 2)) + " promila alkohola!!!");
    }
    catch
    { MessageBox.Show("Ni vseh podatkov, ali pa so podatki napačni!"); }
}

/*metoda za kontrolo vnosa številke: izvede se ob dogodku Validating obeh
 TextBox-ov in gradnika ComboBox na obrazcu*/
private void tB_Validating(object sender, CancelEventArgs e)
{
```

```
    if (sender == cBVrsta)
    {
        /*če uporabnik ni izbral vrste pijače, je indeks -1 (privzeta
          vrednost), se pojavi ikona z opozorilom*/
        if (cBVrsta.SelectedIndex == -1)
            errorProvider1.SetError(cBVrsta, "Izberi vrsto pijače!");
        else /*odstranimo ikono z opozorilom*/
           errorProvider1.SetError((cBVrsta), "");
    }
    else if (sender is TextBox)
    {
        if ((sender as TextBox).Text == "")
             errorProvider1.SetError((sender as TextBox), "Vnesi številko
večjo od 0!");
        else /*odstranimo ikono z opozorilom*/
            errorProvider1.SetError((sender as TextBox), "");
    }
}
```



Slika 60: Obvestilo o napačnem vnosu!



Slika 61: Rezultat pravilnega vnosa.

Dogodki, ki jih sproži miška

Pri obdelavi dogodkov, ki jih proži miška, nas običajno zanima položaj miškinega kazalca in stanje njenih gumbov.

Ko uporabnik premakne miško, operacijski sistem poskrbi, da se miškin kazalec na zaslonu premakne na ustrezno mesto. Kazalec predstavlja eno samo točko (*pixel*), ki ji operacijski sistem sledi in prepoznava kot položaj tega kazalca. Ob premiku miške ali pa pritisku gumba se zgodi eden od dogodkov miške. Trenutni položaj miške lahko tako npr. ugotovimo s pomočjo lastnosti *Location* parametra tipa *MouseEventArgs* (dobimo relativen položaj miške, to je položaj miške znotraj gradnika, v katerem se nahajamo), ali pa npr. z uporabo lastnosti *Position* razreda *Cursor* (dobimo absoluten položaj miške, to je položaj glede na celoten obrazec).

```
/*Pozicija kazalca miške (relativen, glede na gradnik) z uporabo parametra e
tipa MouseEventArgs*/
MessageBox.Show("Lokacija kazalca miške: x = " + e.Location.X.ToString() + ",
y= " + e.Location.Y.ToString());
/*Pozicija kazalca miške (absolutno, glede na obrazec) z uporabo razreda
Cursor*/

MessageBox.Show("Lokacija kazalca miške: x =
"+Cursor.Position.X.ToString()+", y= "+Cursor.Position.Y.ToString());
```

S pomočjo informacije, ki jo dobimo o položaju miške tako lahko izvedemo točno določeno operacijo, ki je odvisna od tega, kje in kdaj smo kliknili z miško.

Temeljni problem, kako obdelati neko potezo narejeno z miško, je pravilna obdelava miškinih dogodkov. V naslednji tabeli so prikazani vsi dogodki, ki jih miška proži, zraven pa je razlaga, kdaj do njih pride.

Dogodek Miške	Razlaga
Click	Dogodek se zgodi ko kliknemo miškin gumb To je tik preden se zgodi dogodek *MouseUp*. Ta dogodek obdelamo tipično tedaj, ko nas zanima samo uporabnikov klik na določen gradnik. Če je gradnik izbran (ima fokus), se kot klik na ta gradnik šteje tudi pritisk na tipko <*Enter*>.
MouseClick	Dogodek se zgodi, ko uporabnik z miško klikne nek gradnik, a ne tudi ob pritisku na tipko <*Enter*> . Ta dogodek uporabimo tedaj, ko želimo ob kliku na nek gradnik dobiti podatke o sami miški. Dogodek *MouseClick* se zgodi za dogodkom *Click*.
DoubleClick	Dogodek se zgodi, ko na nek gradnik dvokliknemo. Uporabimo ga le tedaj, ko res želimo preveriti, ali je uporabnik dvokliknil nek gradnik.
MouseDoubleClick	Dogodek se zgodi, ko uporabnik z miško dvoklikne na določen gradnik. Ta dogodek uporabimo tedaj, ko želimo ob dvokliku na nek gradnik dobiti podatke o sami miški.
MouseDown	Dogodek se zgodi, ko je kazalnik miške nad določenim gradnikom in uporabnik pritisne enega od miškinih gumbov.
MouseEnter	Dogodek se zgodi, ko miškin kazalec doseže rob nekega gradnika, oziroma samo površino tega gradnika – odvisno od vrste gradnika.
MouseHoover	Dogodek se zgodi, ko se z miško zaustavimo in za trenutek obmirujemo nad nekim gradnikom.
MouseLeave	Dogodek se zgodi, ko miškin kazalec zapusti rob določenega gradnika, oz. ko miškin kazalec zapusti površino nekega gradnika – odvisno od vrste gradnika.
MouseMove	Dogodek se zgodi, ko se kazalnik miške, medtem ko smo nad nekom

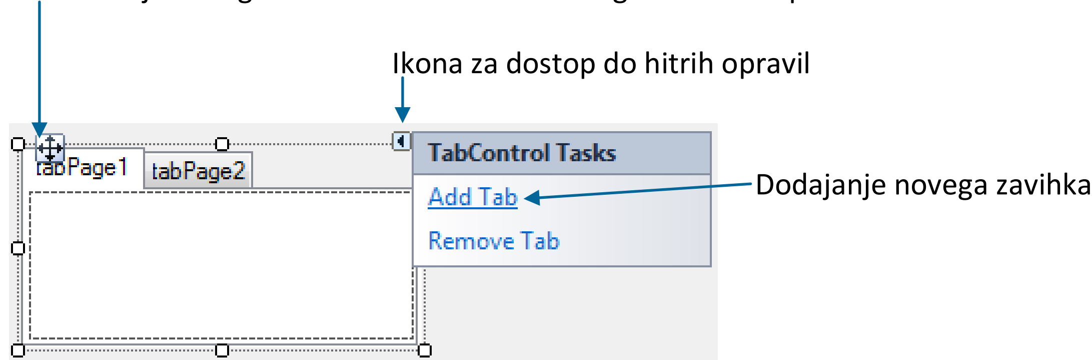
	gradnikom, premakne.
MouseUp	Dogodek se zgodi, ko je kazalec miške nad določenim gradnikom in uporabnik spusti miškin gumb.
MouseWheel	Dogodek se zgodi, ko uporabnik zavrti miškin kolešček, medtem ko je izbran ustrezen gradnik (ima fokus).

Tabela 20: Dogodki miške.

Za vajo ustvarimo nov projekt *DogodkiMiške*, v katerem bomo prikazali dogodke miške. Spoznali bomo tudi uporabo zavihkov in poenostavljen prikazovalnik slik. Za potrebe vaje v oknu *Code View* najprej definirajmo tabelo nekaterih barv in ustvarimo generator naključnih števil

```
Random naklj = new Random();
//tabela barv
Color[] Barve = {
Color.Chocolate,Color.YellowGreen,Color.Olive,Color.DarkKhaki,Color.Sienna,Co
lor.PaleGoldenrod,Color.Peru,Color.Tan,Color.Khaki,Color.DarkGoldenrod,Color.
Maroon,Color.OliveDrab, Color.DarkOrchid };
```

Na obrazec postavimo gradnik *TabControl*: ta gradnik omogoča ustvarjanje poljubnega števila zavihkov. Vsak zavihek lahko vsebuje svoje gradnike, ki so povsem neodvisni od gradnikov na ostalih zavihkih. Gradniku *TabControl* najprej nastavimo lastnost *Dock* na *Fill*, da zapolni celoten obrazec. Zavihke nato tvorimo tako, da kliknemo na ikono v zgornjem desnem delu gradnika in izberemo *Add Tab*.

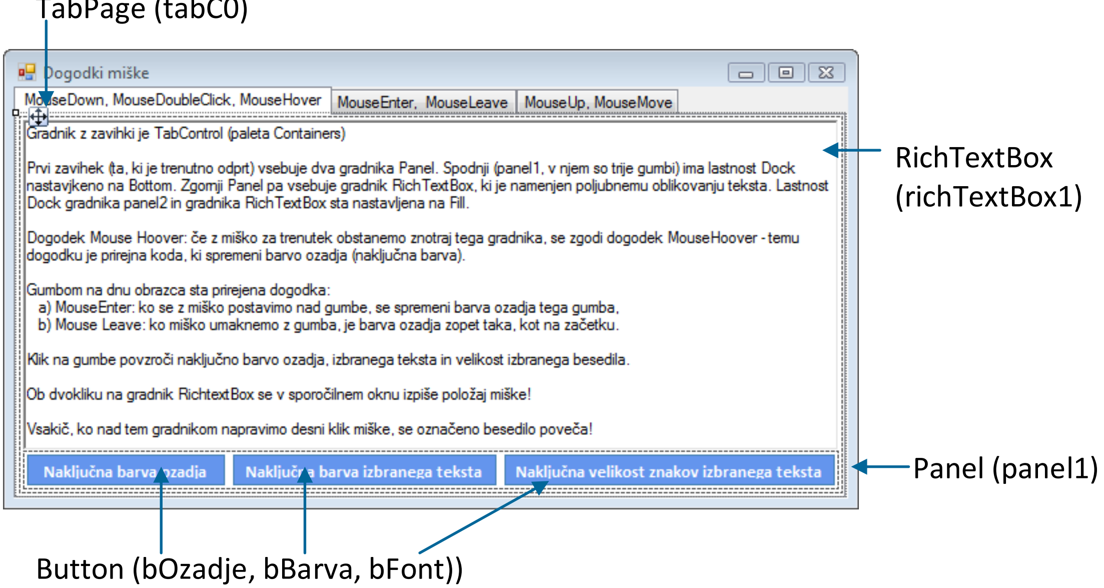


Slika 62: Ustvarjanje novega zavihka gradnika TabControl.

Dva zavihka sta izdelana že ko postavimo gradnik na obrazec, mi pa ustvarimo še enega. Celoten gradnik poimenujmo *tBDogodkiMiske*, posamezne zavihke pa *tabC0*, *tabC1* in *tabC2*. Najprej izberemo posamezen zavihek (kliknemo na zavihek), nato kliknemo na površino tega zavihka in končno glede na zavihek zapišemo v lastnost (*Name)* ime tega zavihka. Posameznim zavihkom določimo še lastnost *Text* tako, kot kaže slika. Ta lastnost določa, kaj je napisano na zavihku. Napisi na zavihkih predstavljajo dogodke miške, ki jih želimo predstaviti na posameznem zavihku. Na površino prvega zavihka postavimo gradnik *Panel* (*panel1*, lastnost

Dock je *Bottom*), nato pa še gradnik *RichTextBox* (lastnost *Dock* je *Fill*). Kliknimo v lastnost *Text* tega gradnika in zapišimo besedilo (komentar), tako kot je na sliki. Gradnik *RichTextBox* je namenjen pisanju teksta, ki ga lahko poljubno oblikujemo.

V gradnik *Panel*, ki je na dnu prvega zavihka postavimo še tri gumbe z imeni *bOzadje*, *bBarva* in *bFont*. Vsem trem gumbom priredimo lastnost *BackColor→CornFlowerBlue*, *ForeColor→*White, ime fonta naj bo *Calibri* velikosti 10, lastnost *FlatStyle* pa naj bo *Flat*. Ta zadnja opredeljuje izgled gumba.



Slika 63: Objektii na prvem zavihku gradnika *TabControl*.

Gumbom zaporedoma priredimo ista odzivni metodi *bOzadje_MouseEnter* in *bOzadje_MouseLeave* (vsem tem gumbom se bo spremenilo ozadje, če bomo vanje vstopili ali izstopili z miško). Spomnimo se, da smo pri projektu *Preverjanje znanja osnovnih računskih operacij* razložili, kako lahko isto odzivno metodo uporabimo nad dogodki različnih gradnikov. Gumbom nato zaporedoma priredimo še odzivne metode *bOzadje_Click*, *bBarva_Click* in *bFont_ Click*.

```
private void bOzadje_MouseEnter(object sender, EventArgs e)
{
      /*ozadje gumba dobi novo barvo*/
      (sender as Button).BackColor = Color.Orange;
}
private void bOzadje_MouseLeave(object sender, EventArgs e)
{
     /*ozadje gumba dobi novo barvo*/
     (sender as Button).BackColor = Color.CornflowerBlue;
}
private void bOzadje_Click(object sender, EventArgs e)
{
    //barva ozadja bo ena izmed barv iz tabele Barve
    richTextBox1.BackColor = Barve[naklj.Next(0, Barve.Length)];
}
```

```
private void bBarva_Click(object sender, EventArgs e)
{
    //tri naključna cela števila med 0 in 254 za mešanje barv
    byte r = Convert.ToByte(naklj.Next(0, 255));
    byte g = Convert.ToByte(naklj.Next(0, 255));
    byte b = Convert.ToByte(naklj.Next(0, 255));
    /*naključno barvo besedila dosežemo z metodo FromArgb razreda Color*/
    richTextBox1.SelectionColor = Color.FromArgb(r, g, b);
}

private void bFont_Click(object sender, EventArgs e)
{
    /*naključno število tipa float med 10 in 20*/
    float velikost=(float)(naklj.NextDouble()*10+10);
    /*ustvarimo nov objekt tipa Font: uporabimo enega izmed konstruktorjev
      razreda Font: parametri konstruktorja so ime fonta, velikost in stil */
    Font novi = new Font(richTextBox1.Font.Name, velikost, FontStyle.Bold);
    /*richTextBox1.Font = novi;  //Takale bi bila npr.nastavitev novega fonta
      za celoten RichTextBox1*/
    richTextBox1.SelectionFont = novi;//izbrani tekst dobi nov font
}
```

Gradniku *richTextBox1* priredimo odzivne metode za dogodke *MouseDown*, *MouseDoubleClick* in *MouseHover* (za dogodek MouseHover v oknu *Properties*→*Events*→MouseHover izberemo že prej napisano odzivno metodo *bOzadje_Click*).

```
private void richTextBox1_MouseDown(object sender, MouseEventArgs e)
{
    /*preverimo, če je bil pritisnjen desni gumb*/
    if (e.Button == MouseButtons.Right)
    {
         /*označeno besedilo gradnika richTextBox1 se poveča za ena*/
         richTextBox1.SelectionFont = new
Font(richTextBox1.SelectionFont.Name, richTextBox1.SelectionFont.Size + 1,
FontStyle.Italic);
    }
}

private void richTextBox1_MouseDoubleClick(object sender, MouseEventArgs e)
{
    MessageBox.Show("Položaj oz. koordinate miške\n\nDvokliknjena je bila
tipka : "+e.Button+"\nOddaljenost od levega roba: " + e.X + "\nOddaljenost od
vrha obrazca: " + e.Y);
}
```

Preklopimo sedaj na drugi zavihek. Kot smo že napovedali, bomo na tem zavihku delali slikami. Potrebujemo torej prostor za 9 slik.

Na zavihek najprej postavimo gradnik *Panel* in ga poimenujmo *panel2*. Ker mu lastnost *Dock* nastavimo na *Left*, bomo s tem dosegli, da bo gradnik prilepljen na levem delu zavihka. V ta gradnik nato postavimo 9 enakih gradnikov *PictureBox* z imeni *picureBox1* do *pictureBox9*. Vsi

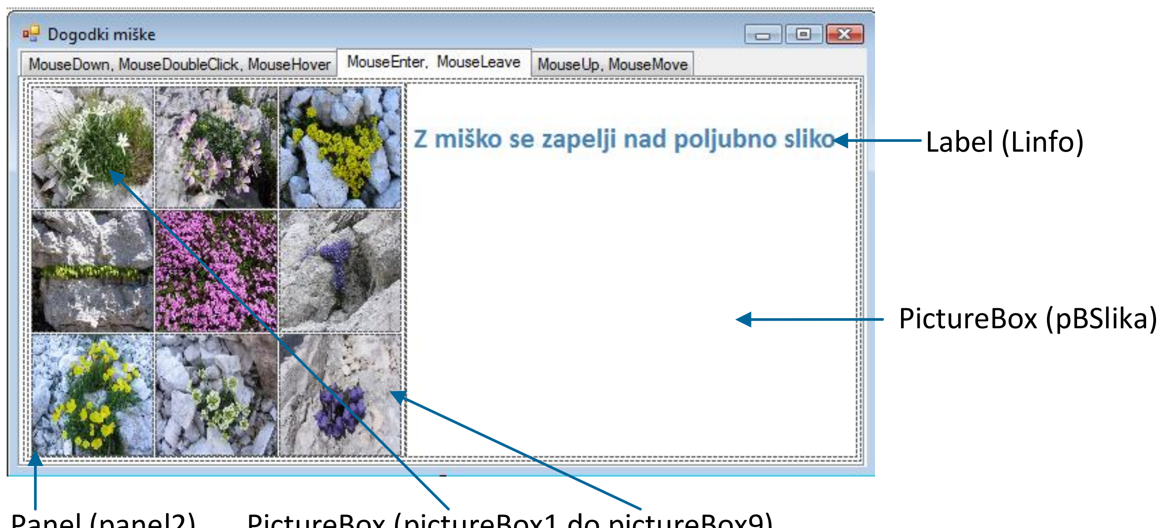
imajo lastnost *SizeMode* nastavljeno na *StretchImage*. S tem dosežemo, da bo v gradniku prikazana, povečana ali pa pomanjšana, celotna slika, ne glede na to, če bo uporabnik morda spremenil velikost okna. Velikost gradnika naj bi npr. 90 x 90. Kljub temu velja omeniti, da je priporočljivo, da si vsaj v osnovi pripravimo slike, ki so ustrezne velikosti. Programi za delo s slikami namreč povečevanje/pomanjševanje slik opravijo praviloma kvalitetneje, kot so metode, vgrajene v C#, saj gre za specializirane programe, namenjene prav temu.

Na desno stran drugega zavihka zatem postavimo še en gradnik *PictureBox* in ga poimenujmo *pBSlika*, lastnost *Dock* naj bo *Fill*, lastnost *SizeMode* pa naj bo zopet *StretchImage*. V gradnike *pictureBox1* do *pictureBox9* sedaj uvozimo devet različnih slik. Končno v gradnik *pictureBox9* postavimo še gradnik *Label* z napisom kot je na sliki.

Gradnikom *pictureBox1* do *pictureBox9* priredimo isti odzivni metodi dogodkov *MouseEnter* in *MouseLeave*. Namen teh dveh metod je, da ko se z miško postavimo na katerokoli sliko, se le-ta povečana prikaže v desnem delu zavihka (obenem pa se skrije oznaka z napisom), ko pa z miško sliko zapustimo, je desna stran zavihka zopet taka kot prej (oznaka z napisom se zopet prikaže).

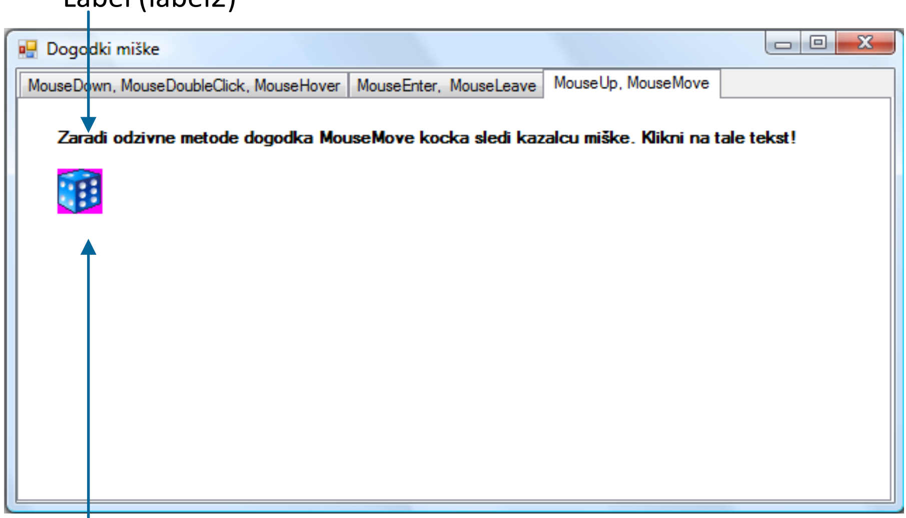
```
private void pictureBox1_MouseEnter(object sender, EventArgs e)
{
    lInfo.Visible=false;
    pbSlika.Image = (sender as PictureBox).Image;
}

private void pictureBox1_MouseLeave(object sender, EventArgs e)
{
    pbSlika.Image = null;
    lInfo.Visible = true;
}
```

Slika drugega zavihka:



Slika 64: Objekti na drugem zavihku gradnika *TabControl*.

Preklopimo še na tretji zavihek. Nanj postavimo le dva gradnika: gradnik *Label* in gradnik *PictureBox* (lastnost *SizeMode* naj bo *StretchImage*).



Slika 65: Objekta na tretjem zavihku gradnika *TabControl*.

Gradniku *tabC2* (tretjemu zavihku) priredimo odzivno metodo dogodka *MouseMove* in ga poimenujmo *tm_MouseMove*, gradniku *Label* pa dogodek *MouseUp*. S pomočjo dogodka *MouseMove* in parametrom tipa *MouseEventArgs*, ki vsebuje tudi lastnosti *X* in *Y* (to sta koordinati miške) smo dosegli potovanje slike skupaj z miškinim kazalcem.

```
private void label2_MouseUp(object sender, MouseEventArgs e)
{
    /*fontu oznake priredimo nov objekt tipa Font: uporabimo enega izmed
      konstruktorjev razreda Font: s parametri konstruktorja povemo, v katero
      družino ta font spada, velikost (tipa Float) in stil */
    label2.Font = new Font(FontFamily.GenericSansSerif,20.0F,FontStyle.Bold);
    label2.Text = "Ko si spustil tipko, se je zgodil dogodek
        \r\nMouseUp\r\n\r\nTrenutni čas: "+DateTime.Now.ToLongTimeString();
}

private void tm_MouseMove(object sender, MouseEventArgs e)
{
    //ob premiku miške, bo slika sledila miški!
    pictureBox10.Location = new Point(e.X, e.Y);
}
```

Point v C# je razred, ki predstavlja točko z njenima koordinatama v dvodimenzionalnem koordinatnem sistemu. Konstruktorju objekta smo posredovali dve koordinati, ki predstavljata trenutni položaj miškinega kazalca. Ta nova točka tako postane novi položaj ikone kocke v gradniku *PictureBox*.

Labirint

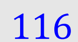
Izdelajmo preprosto igro *Labirint*. Uporabnik bo moral kazalnik miške premakniti od levega zgornjega roba do spodnjega desnega roba obrazca, pri čemer ne sme zadeti stene labirinta. Zamislili smo si, da bo igra videti tako, kot prikazuje *Slika 66: Sestavni deli labirinta*. Na sliki smo že označili gradnike, ki bodo potrebni. Kot vidimo, tokrat izjemoma ne bomo potrebovali nobenega novega.

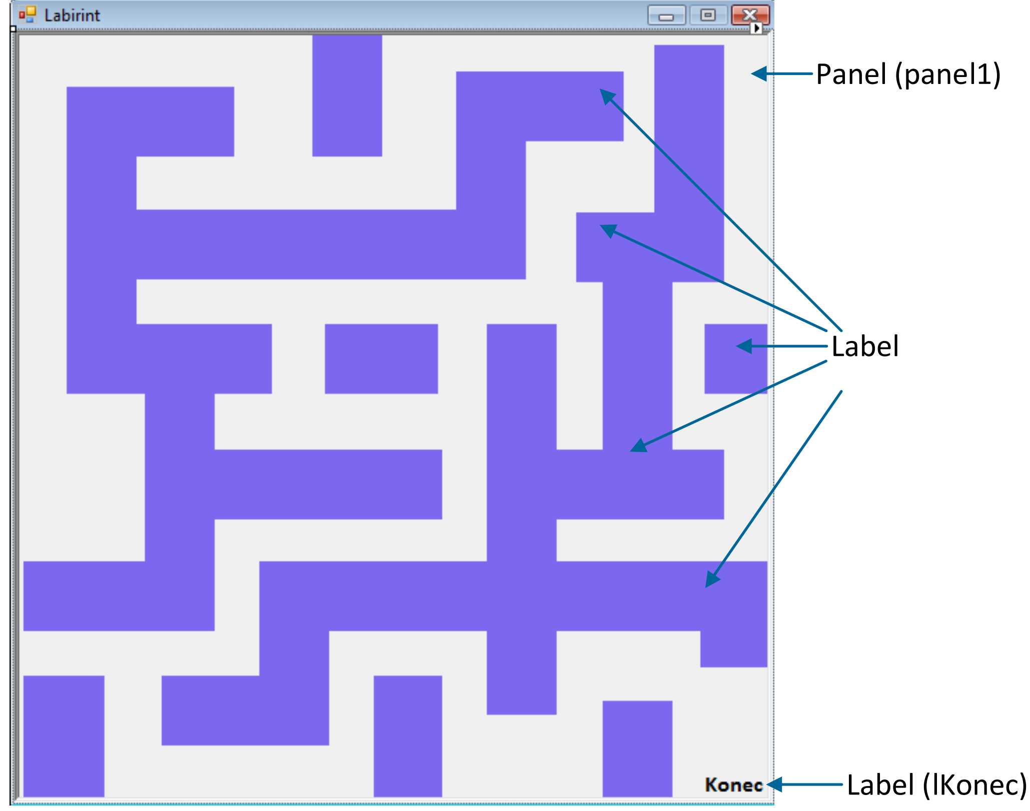
Lastnost *Text* obrazca zapišimo *Labirint* in obrazcu določimo velikost, kakršno menimo, da bomo potrebovali za naš labirint. Ker bomo lastnost *FormBorderStyle* nastavili na *Fixed3D* in lastnost *MaximazeBox* na *False*, uporabnik velikosti okna ne bo mogel spreminjati, niti ga maksimirati.

Na obrazec postavimo gradnik *Panel* in mu lastnost *Dock* nastavimo na *Fill*. S tem ga raztegnemo čez celotno površino. Ker mu lastnost *BorderStyle* nastavimo na *Fixed3D*, bodo robovi obrazca dobro vidni.

Določiti moramo še, kje je izhod iz labirinta. V ta namen na obrazec postavimo še gradnik *Label*. Poimenujmo (lastnost *(name)*) ga kar *lKonec* in mu lastnost *Text* spremenimo v *Konec*. Oznako premaknimo v spodnji desni rob obrazca, vendar se ne sme dotikati njegovih robov.

Labirint moramo še ustrezno oblikovati. V ta namen si pripravimo gradnik tipa *Label*, ki ga bomo oblikovali v obliko polnega pravokotnika. Zato mu pobrišimo lastnost *Text* in ga pobarvamo tako, da mu določimo lastnost *BackColor* (v vaji je to *MediumStatBlue*). Ko mu poskušamo s pomočjo vlečenja spremeniti velikost, vidimo, da ne gre. Razlog je v tem, da je lastnost *AutoSize* privzeto nastavljena na *True*. To pomeni, da se velikost oznake avtomatsko spreminja glede na to, koliko prostora potrebujemo za prikaz besedila na oznaki (lastnost *Text*). Ko mu lastnost *AutoSize* nastavimo na *False*, našo "opeko" lahko nastavimo na poljubno velikost.





Slika 66: Sestavni deli labirinta.

S pomočjo tehnike *Copy – Paste* iz teh oznak oblikujmo svoj labirint, pri čemer posamezne oznake poljubno povečujemo oz. jih pomanjšujemo. Paziti moramo le na to, da bo prazen prostor med posameznimi oznakami dovolj velik, da ne bo igra pretežka. Seveda pa moramo poskrbeti tudi za to, da v labirintu obstaja vsaj ena pot od začetka do konca.

Ostane nam le še zapis ustreznih dogodkov. Uporabnik bo moral s kazalnikom miške priti od začetka labirinta (v zgornjem levem kotu obrazca) do konca. Najprej napišimo metodo *PremikNaZacetek*, ki bo kazalnik miške na začetku postavila v zgornji levi kot obrazca. Pred pisanjem metode ustvarimo še dva objekta tipa *SoundPlayer*: kadarkoli bomo zadeli opeko, se bo oglasil zvočnik z nekaj toni *"chord.wav"*. Ob uspešnem zaključku igre pa se bo oglasil zvok *"tada.wav"*. Več o tem gradniku smo si že ogledali v projektu *Predvajalnik glasbe in videa*.

```
//Objekt tipa SoundPlayer, ki se oglasi vselej, ko igralec zadene ob opeko
System.Media.SoundPlayer zadetekOpeke = new
System.Media.SoundPlayer(@"C:\Windows\Media\chord.wav");
//Objekt tipa SoundPlayer, ki se oglasi ko igralec uspešno zaključi igro
System.Media.SoundPlayer zakljucniZvok = new
System.Media.SoundPlayer(@"C:\Windows\Media\tada.wav");
```

Sedaj pripravimo metodo, ki bo povzročila premik kazalca miške na mesto, ki je 10 točk oddaljeno od zgornjega roba in 10 točk od levega roba.

```
private void PremikNaZacetek()
```

```
{
    zadetekOpeke.Play();//Začetni zvok
    /*Objekt tipa Point v C# predstavlja že narejen razred, ki predstavlja
      točko z dvema koordinatama v dvodimenzionalnem koordinatnem sistemu*/
    Point zacetnaTocka = panel1.Location;
    zacetnaTocka.Offset(10, 10); //Premik točke
    Cursor.Position = PointToScreen(zacetnaTocka); //Začetni položaj miške
}
```

S *panel1.Location* smo ugotovili, kje na zaslonu je levi zgornji vogal panela, kjer je labirint. Z metodo *Offset* smo potem to točko premaknili 10 točk desno in 10 točk dol, saj je koordinatni sistem tak, da koordinate x naraščajo v desno, koordinate y pa navzdol.

S *Cursor.Position* nadziramo položaj miške. Metoda *PointToScreen* izračuna koordinate točke, torej ravno točko, kamor želomo postaviti miškin kazalec.

Metodo *PremikNaZacetek()* bomo poklicali pri zagonu projekta (v konstruktorju obrazca)

```
public Form1()
{
    InitializeComponent();
    //Klic metode: začetna pozicija miške
    PremikNaZacetek();
}
```

in vselej, ko uporabnik zadene ob neko opeko. Zadetek opeke je dogodek *MouseEnter*, ki ga priredimo vsem opekam na obrazcu (vsem oznakam). To storimo tako, da najprej izberemo poljubno opeko, v oknu *Properties→Events* v polje *MouseEnter* zapišemo ime dogodka *Opeka_MouseEnter*, nato dvokliknemo na to ime in zapišemo ustrezno kodo – to bo le klic že napisane metode *PremikNaZacetek*.

```
private void Opeka_MouseEnter(object sender, EventArgs e)
{
    /*Ko kazalec miške zadene opeko ali pa vstopi v gradnik panel,
      pokličemo metodo PremikNaZacetek()*/
    PremikNaZacetek();
}
```

Dogodek *Opeka_MouseEnter* moramo sedaj prirediti vsem opekam na obrazcu. Najlažje tako, da v meniju *Edit* razvojnega okolja kliknemo *Select All*. S tem smo izbrali vse objekte, tudi oznako *lKonec*, ki pa je ne želimo imeti med izbranimi. Kot običajno v okolju *Windows* stisnemo tipko *Ctrl* in kliknemo na oznako z napisom *Konec*. Tej oznaki bomo kasneje priredili dogodek za zaključek igre. Izbrane imamo torej vse gradnike, razen oznake z napisom *Konec*. V oknu *Properties→Events* v polju *MouseEnter* sedaj izberemo dogodek *Opeka_MouseEnter*, ki smo napisali že prej. Priredi se vsem izbranim objektom.

Ostane še dogodek *MouseEnter* oznake *lKonec*. V sporočilnem oknu igralcu čestitamo za uspešen zaključek, še prej pa naj se oglasi zvočnik s fanfarami!

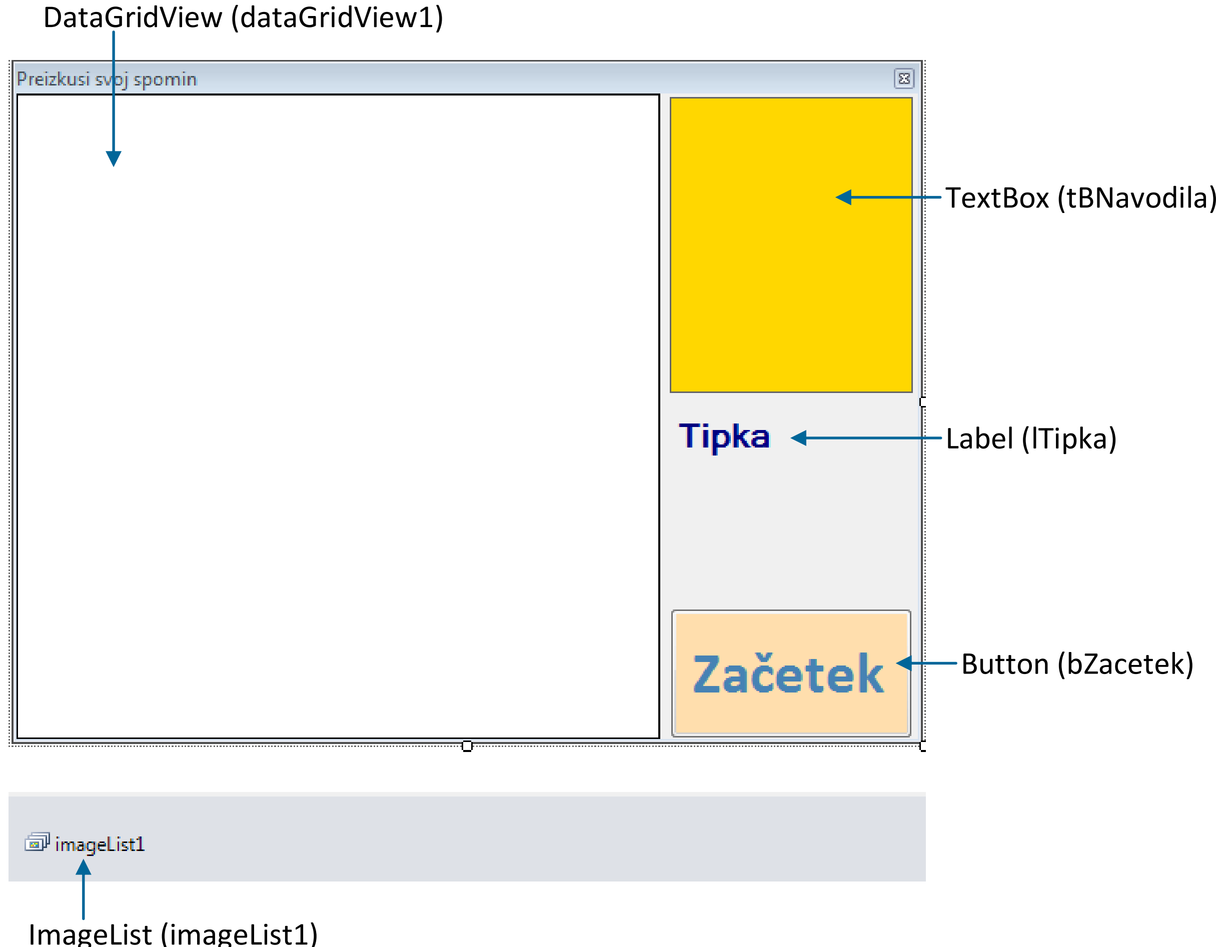
```
private void lKonec_MouseEnter(object sender, EventArgs e)
```

```
{
    //fanfare na koncu in še obvestilo o zaključku igre
    zakljucniZvok.Play();
    MessageBox.Show("Čestitke!");
    Close();
}
```

Spomin – igra za preizkus spomina

Spomnimo se igre, ki smo se je igrali v zgodnji mladosti. Na voljo imamo 16 podvojenih sličic. Na začetku so sličice razporejene naključno in obrnjene narobe. V čim manj korakih skušamo odkriti pare tako, da obrnemo po dve sličici. Če sta enaki, ju pustimo odkriti, sicer pa ju obrnemo nazaj. Za to igro napišimo ustrezen program, v katerem bomo uporabili gradnik *DataGridView*.

V tem projektu bomo vrstice s podatki (slike) dodajali in spreminjali programsko. Pokazali bomo, kako dostopamo do posamezne vrstice in posamezne celice tega gradnika. Slike, ki jih bomo potrebovali, bomo hranili v gradniku tipa *ImageList*, da bomo do njih lahko dostopali preko indeksa. Ko gradnik *ImageList* postavimo na obrazec, se ta umesti na dno urejevalniškega okna, pod obrazcem. Gre namreč za nevizuelni gradnik, ki ga na obrazcu neposredno ne vidimo, saj je njegov namen le hranjenje slik. Slike vanj dodamo tako, da ga izberemo, nato pa v zgornjem desnem kotu kliknemo na puščico. V oknu, ki se odpre izberemo velikost slik (*ImageSize*) npr. 256 x 256, nato pa kliknemo na *Choose images*: v pogovornem oknu (*Image Collecton Editor*), ki se pri tem odpre, nato s klikom na gumb *Add* v ta seznam uvozimo 8 različnih slik, ki jih bomo uporabili v gradniku *DataGridView*, deveto sliko (njen indeks bo 8), pa pripravimo v poljubnem grafičnem urejevalniku (lahko kar v *Slikarju*). Ta slika naj bo le ustrezne velikosti (256 x 256) in naj ima le belo ozadje – nič drugega! Uporabili jo bomo za začetne slike celic gradnika *DataGridView* . Načrt uporabniškega vmesnika bo sledeč:



Slika 67: Gradniki na obrazcu *FSpomin*.

V vnosnem polju *tBNavodila* bomo prikazali osnovna navodila. Slike bomo odkrivali s klikom miške, z oznako *lTipka* pa bomo uporabniku sporočali, katera tipka miške je na vrsti. Na obrazcu je še gumb, s katerim bomo igro zagnali.

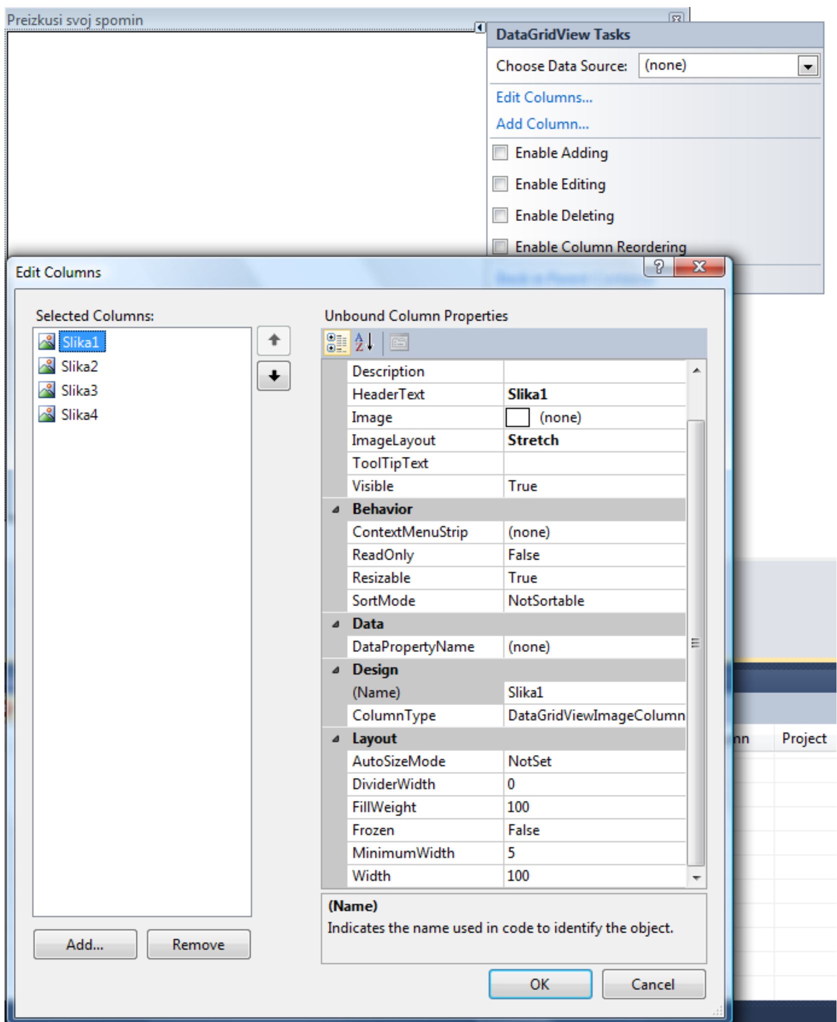
Lastnosti gradnikov na obrazcu so prikazane v naslednji tabeli:

Gradnik	Lastnost	Nastavitev	Opis
Form1	FormBorderStyle	FixedToolWindow	Uporabnik velikosti okna ne more spreminjati
	Name	FSpomin	Ime obrazca
tBNavodila	BackColor	Gold	Barva ozadja
	BorderStyle	FixedSingle	Okvir okoli besedila
	MultiLine	True	Večvrstično besedilo

	ForeColor	Indigo	Barva besedila
bZacetek	BackColor	NavajoWhite	Barva ozadja
		StellBlue	Barva besedila
dataGridView1	BackGroundColor	White	Barva ozadja
	AllowUserToAddRows	False	Uporabnik ne more dodajati vrstic
	AllowUserToDeleteRows	False	Uporabnik ne more brisati vrstic
	AllowUserToResizeColumns	False	Uporabnik ne more spreminjati velikosti vrstic
	AllowUserToResizeRows	False	Uporabnik ne more spreminjati velikosti stolpcev
	ColumnHeadersVisible	False	Imena stolpcev niso vidna
	Columns	S klikom na gumb *Add* dodamo 4 stolpce: stolpcem priredimo lastnost *Name* in *HeaderText* takole: *Slika1*, *Slika2*, *Slika3* in *Slika4*, za lastnost *Type* pa izbremo *DataGRidViewImageColumn* (stolpec s sliko). Vsem stolpcem določimo še lastnost *ImageLayout – Stretch*. Na ta način se bo velikost slike avtomatsko prilagodila velikosti celice gradnika *DataGridView*.	
	Dock	Left	Gradnik bo zapolnil celotno levo stran obrazca
	Enabled	False	Gradnik na začetku ni aktiven
	RowHeadersVisible	False	Glave vrstic niso vidne
	SCrollBars	Vertical	Navpični drsnik

Tabela 21: Lastnosti gradnikov obrazca *FSpomin*.

Izdelavo in poimenovanje stolpcev gradnika *DataGridView* , ter določanje njihovih lastnosti prikazuje naslednje slika. Vsi stolpci imajo enake lastnosti, le imena so različna. Zato tudi tu nastavljanje hitreje opravimo tako, da s pomočjo tipke *Ctrl* izberemo vse hkrati in jim nastavimo skupne lastnosti.



Slika 68: Izdelava stolpcev v gradniku *DataGridView* in nastavitev lastnosti.

V primeru, da smo sliko postavili v datoteko virov (kako to storimo smo pokazali v projektu *Predvajalnik glasbe in videa*), pa smo to storili pomotoma oz. jo želimo od tam odstraniti, to na tem mestu ni mogoče – odstranimo jo tako, da v *Solution Explorerju* poiščemo datoteko *Resources.resx*. Dvokliknemo na to datoteko in prikaže se nam njena vsebina - odstranimo vse nezaželene vnose.

Slike v gradniku *DataGridView* bomo prikazovali z desnim klikom miške, ugibali pa jih bomo z levim klikom. Za naključno razporeditev slik bomo poskrbeli s pomočjo tabele celih števil, katere vsebina bo predstavljala indekse slik, ki bodo v posameznih celicah gradnika *DataGridView*. Potrebujemo tudi spremenljivki tipa *DateTime*, s pomočjo katerih bomo merili čas, ki ga bo tekmovalec potreboval za uspešno opravljeno nalogo. V konstruktrju obrazca poskrbimo še za začetni izgled igre, dodajmo pa še dve odzivni metodi, s katerima spreminjamo izgled gumba za začetek igre.

```
Public partial class Fspomin : Form
{
    int[] tabStevil;//deklaracija tabele naključnih slik
    bool prikaziSliko;//indikator, da je na vrsti nova slika in ne ugibanje
    int indeksDrugeSlike; //indeks druge slike v tabeli celih števil
```

```
    int vPrva, sPrva;//številka vrstice in stolpca za prvo sliko
    int zadetihParov;//število zadetih parov
    Random naklj=new Random();//generator naključnih števil
    DateTime zacetek, konec;
    public Fspomin()
    {
        InitializeComponent();
        //položaj okna bo na začetku na sredini zaslona
        this.StartPosition = FormStartPosition.CenterScreen;
    }
    private void Form1_Load(object sender, EventArgs e)
    {
        this.Height = 427;//začetna širina obrazca
        //začetni stil obrazca
        this.FormBorderStyle = FormBorderStyle.FixedToolWindow;
        dataGridView1.Width = 402;//začetna širina gradnika DataGridView
        //začetna višina stolpca gradnika DataGridView
        dataGridView1.RowTemplate.Height = 100;
        lTipka.Text = "";//skrijem oznako za oznako tipke
        //vse slike so nazačetku bele: bela slika ima indeks 8
        for (int i=0;i<4;i++) //nove vrstice z belimi slikami (ozadjem)
            dataGridView1.Rows.Add(imageList1.Images[8],
imageList1.Images[8], imageList1.Images[8], imageList1.Images[8]);
        tBNavodila.Text="KRATKA NAVODILA:\r\n\r\nOdkriti moraš 8 parov enakih
slik. Z desno tipko odkriješ prvo sliko, nato pa z levo ugibaš njen
par!\r\n\r\nZa začetek klikni gumb 'Začni'! ";
        tBNavodila.ReadOnly = true;
        /*onemogočimo klikanje na gradnik DataGridView dokler ni kliknjen
         gumb Začetek*/
        dataGridView1.Enabled = false;
    }
    private void bZacetek_MouseEnter(object sender, EventArgs e)
    {
        bZacetek.BackColor = Color.DarkSlateBlue;
        bZacetek.ForeColor = Color.White;
    }
    private void bZacetek_MouseLeave(object sender, EventArgs e)
    {
        bZacetek.BackColor = Color.NavajoWhite;
        bZacetek.ForeColor = Color.SteelBlue;
    }
}
```

Odzivna metoda ob kliku na gumb *Začetek* poskrbi za naključno razporeditev slik.

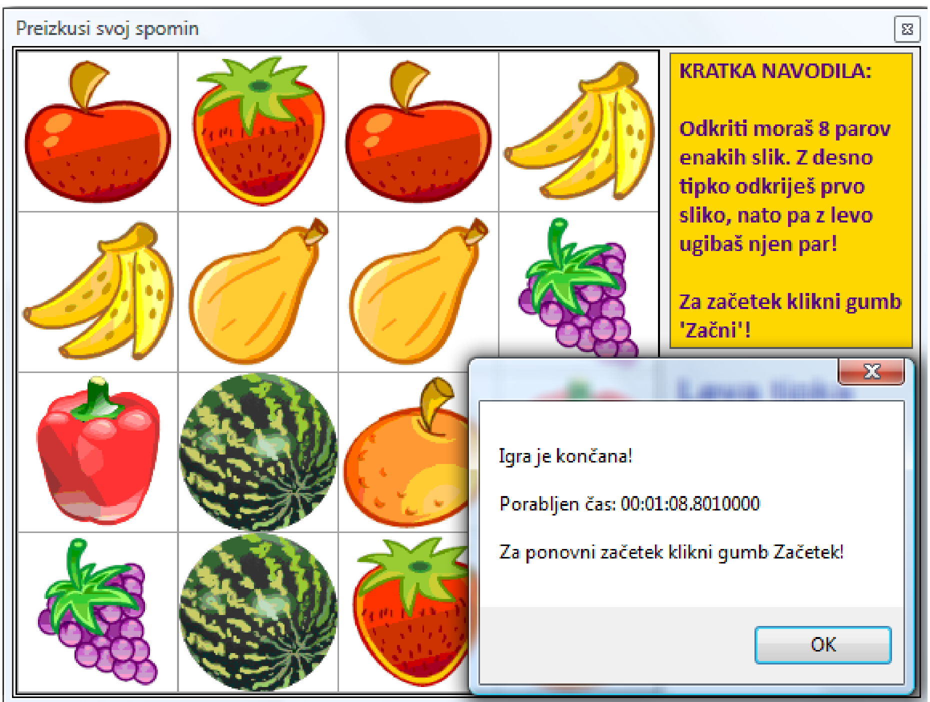
```
private void bZacetek_Click(object sender, EventArgs e)
{
    RazporediSlike();
}
private void RazporediSlike()
{
    prikaziSliko = true;//indikator da je na vrsti izbira slike(desna tipka)
    /*inicializiramo tabelo 16 naključnih števil (število se ponovi 2x)*/
```

```
    tabStevil = new int[16];
    for (int i = 0; i < 16; i++)
        tabStevil[i] = -1; //tabelo napolnimo z števili -1
    for (int i=0;i<8;i++)
    {
        int stCelice1; //določimo številko prve še proste celice
        do
        {
            stCelice1=naklj.Next(16);//naključno celo število med 0 in 15
        } while (tabStevil[stCelice1]!=-1);
        int stevilo;
        do  /*naključno številko med 0 in 7, ki še ni v nobeni celici*/
        {
            stevilo = naklj.Next(8);/*naključno število med vključno 0 in
                                      vključno 7*/
        } while (tabStevil.Contains(stevilo));
        tabStevil[stCelice1] = stevilo;
        int stCelice2;//določimo številko druge še proste celice
        do
        {
           stCelice2 = naklj.Next(16);//naključno celo število med 0 in 15
        } while (tabStevil[stCelice2]!=-1);
        tabStevil[stCelice2]=stevilo;//število zapišemo v tabelo
    }
    dataGridView1.Enabled = true; //klik na gradnik DataGridView je mogoč
    zadetihParov = 0;//število zadetkov je na začetku 0
    lTipka.Text = "Desna tipka";//besedilo na na oznaki
    bZacetek.Visible = false;//skrijemo gumb Začetek
    dataGridView1.Focus();//gradnik DataGridView je aktiven gradnik
    zacetek = DateTime.Now;//trenutni čas
}
```

Slike bomo odkrivali z desnim oziroma levim klikom miške, zato potrebujemo odzivno metodo dogodka *CellMouseClick*.

```
//desni klik v okno prikaže prvo sliko
private void dataGridView1_CellMouseClick(object sender,
DataGridViewCellMouseEventArgs e)
{
    if (e.Button == MouseButtons.Right)//pritisnjen desni gumb miške
    {
        if (prikaziSliko)//če je na vrsti desna tipka
        {
            vPrva = e.RowIndex; //shranimo indeks vrstice za prvo sličico
            sPrva = e.ColumnIndex;//še indeks stolpca za prvo sličico
            dataGridView1.Rows[vPrva].Cells[sPrva].Value =
                 imageList1.Images[(tabStevil[ vPrva * 4 + sPrva])];
            prikaziSliko = false;/*indikator, da je na vrsti ugibanje
                                   slike (leva tipka)*/
            lTipka.Text = "Leva tipka";/*navodilo uporabniku, katero
                                         tipko naj pritisne*/
        }
    }
```

```
    /*če je na vrsti levi gumb in če je ta pritisnjen in slika v celici
       Ni bela (indeks slike je 8) */
    else if (!prikaziSliko && e.Button == MouseButtons.Left &&
              dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex].Value !=
              imageList1.Images[8] &&
dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex] !=
dataGridView1.Rows[vPrva].Cells[sPrva]
)
    {
        //glede na izbiro celice izračunamo indeks druge slike
        indeksDrugeSlike = e.RowIndex * 4 + e.ColumnIndex;
        /*v izbrani celici prikažemo sliko, ki ji ustreza glede na
          številko v tabeli tabStevil*/
        dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex].Value =
          imageList1.Images[(tabStevil[indeksDrugeSlike])];
        //indikator da je na vrsti izbira slike (desna tipka)
        prikaziSliko = true;
        //Preverimo, če sta sliki enaki
        if (tabStevil[vPrva * 4 + sPrva] == tabStevil[indeksDrugeSlike])
        {
            zadetihParov++;//sliki sta enaki, povečamo števec zadetkov
            if (zadetihParov == 8)//če so vsi pari odkriti
            {
                konec = DateTime.Now; //trenutni čas
                MessageBox.Show("Igra je končana!\n\nPorabljen čas: "+(konec-
          zacetek).ToString()+"\n\nZa ponovni začetek klikni gumb Začetek!");
                //Pobrišemo vsebino gradnika DataGridView
                dataGridView1.Rows.Clear();
                for (int i = 0; i < 4; i++)//Vse slike so na začetku bele
                   dataGridView1.Rows.Add(imageList1.Images[8],
     imageList1.Images[8], imageList1.Images[8], imageList1.Images[8]);/*nova
                                                                   vrstica */
                RazporediSlike();//nova razporeditev slik
                bZacetek.Visible = true;//skrijemo gumb
            }
        }
        //če izbrani sličici nista enaki, ju moramo pobisat
        else
        {
            MessageBox.Show("Poskusi znova!","Izbira
               napačna",MessageBoxButtons.OK,MessageBoxIcon.Warning);
            //skrijem prvo sliko: slika je zopet bela
            dataGridView1.Rows[vPrva].Cells[sPrva].Value =
               imageList1.Images[8];
            //skrijemo drugo sliko: slika je zopet bela
            dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex].Value =
               imageList1.Images[8];
            lTipka.Text = "Desna tipka";
        }
    }
}
```



Slika 69: Končni izgled programa *Spomin*!

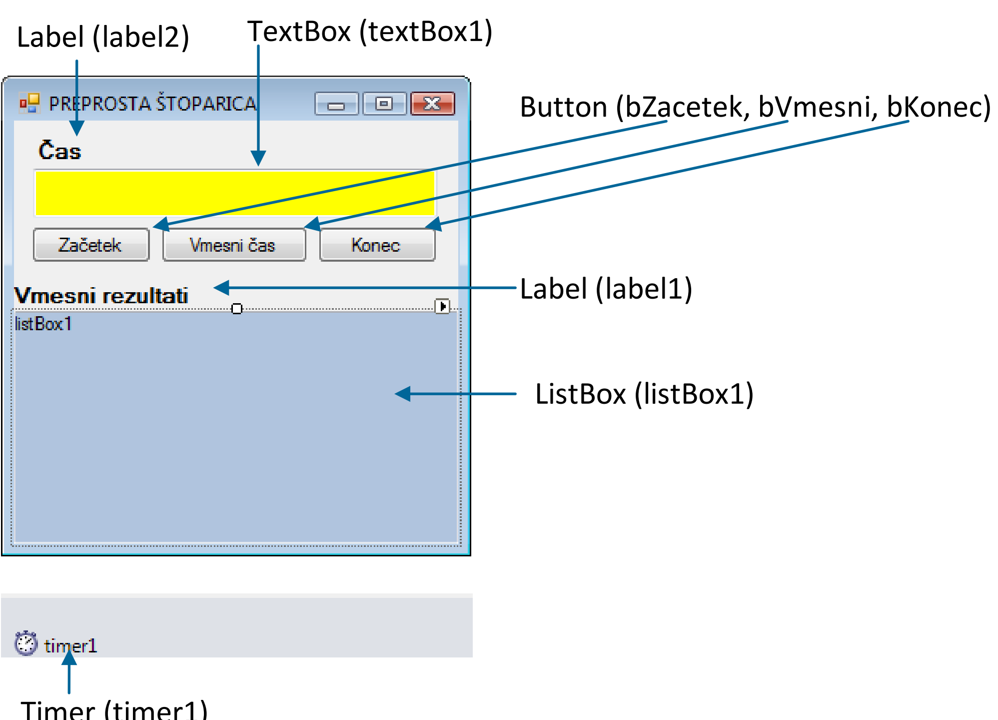
Dogodki, ki jih sproži ura (gradnik *Timer*) in metode za delo s časom

Gradnik *Timer* (programska ura) je namenjen generiranju dogodkov v določenih časovnih intervalih. V oknu *Toolbox* se nahaja v skupini *Components*. Ta gradnik je eden izmed številnih nevizuelnih gradnikov. Ko ga postavimo na obrazec, se avtomatično namesti v posebno polje pod samim obrazcem. Gre pač za gradnik, ki na samem obrazcu ne bo viden, bo pa opravljal neko funkcijo.

Gradnik ima za razliko od ostalih gradnikov le nekaj lastnosti, od katerih sta poleg imena pomembni le dve: to sta lastnosti *Enabled* in *Interval*. Lastnost *Enabled* je privzeto postavljena na *False*. Če hočemo torej gradnik postaviti v operativno funkcijo, mu moramo lastnost *Enabled* postaviti na *True*. Z lastnostjo *Interval* pa nastavimo časovni interval v milisekundah, ki bo pretekel med posameznima dogodkoma, ki jih bo sprožil ta gradnik. Vrednost 1000 npr. pomeni vsako sekundo, vrednost 100 eno desetinko sekunde ...

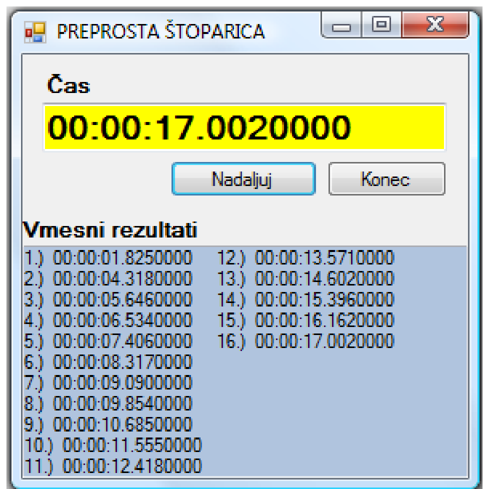
Gradniku *Timer* lahko v oknu *Properties→Events* priredimo odziv na en sam dogodek. To je dogodek *Tick*, ki se zgodi vsakokrat, ko preteče čas, podan z lastnostjo *Interval*.

Zgradimo projekt, v katerem bomo predstavili preprosto štoparico. V projekt moramo vključiti nevizuelni gradnik *Timer*. Izgled obrazca naj bo takle:



Slika 70: Gradniki na obrazcu za prikaz štoparice.

Za merjenje časa bomo uporabili dva objekta tipa *DateTime* z imenoma *Zacetek* in *Konec*. Inicializirali ju bomo ob kliku na gumba *Zacetek* in *Konec*. Klik na gumb *Začetek* predstavlja začetek merjenja časa, ob kliku na gumb *Vmesni čas* se pretečeni čas zapiše v gradnik *ListBox*, obenem pa se prikazovanje časa v gradniku *TextBox* zaustavi za toliko časa, dokler ponovno ne pritisnemo tega gumba. Ob kliku na gumb *Konec* se merjenje časa dokončno zaustavi.



Slika 71: Prikaz delovanja štoparice.

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

```
    }
    DateTime Start; //objekt, ki bo hranil čas začetka merjenja
    DateTime End;   //objekt, ki bo hranil končni čas
    //objekt tipa TimeSpan potrebujemo za merjenje časovnega intervala
    TimeSpan izmerjeniCas;
    private void Form1_Load(object sender, EventArgs e)
    {
        bVmesni.Visible = false;
        bKonec.Visible = false;
    }
}
```

V odzivni metodi dogodka *Load* smo poskrbeli, da sta vmesni in končni čas na začetku nevidna.

Potrebujemo še logično spremenljivko, ki bo med delovanjem programa določala, ali naj čas v gradniku *textBox1* teče, ali pa ne. Odzivna metoda dogodka *Tick* gradnika *Timer* bo poskrbela za prikaz pretečenega časa. Gradniku *Timer* pustimo lastnost *Enabled* kar privzeto, to je *False*.

```
/*spremenljivka casTece določa,ali v gradniku texBox1 čas teče ali ne*/
bool casTece;
int stevec = 1;//števec vmesnih časov
//odzivna metoda timerja
private void timer1_Tick(object sender, EventArgs e)
{
    /*ob vsakem dogodku Tick gradnika Timer se osveži vsebina gradnika
      textBox1*/
    if (casTece)
        textBox1.Text=Convert.ToString(DateTime.Now-Start);
}
```

Napišime še odzivne metode vseh treh gumbov. Ob kliku na gumb *Zacetek* poskrbimo za začetne vrednosti objektov in sprožimo *Timer*. Klik na gumb *Konec* zaustavi *Timer* in prikaže končni čas, vmesne čase pa sporočamo in jih shranjujemo ob kliku na srednji gumb.

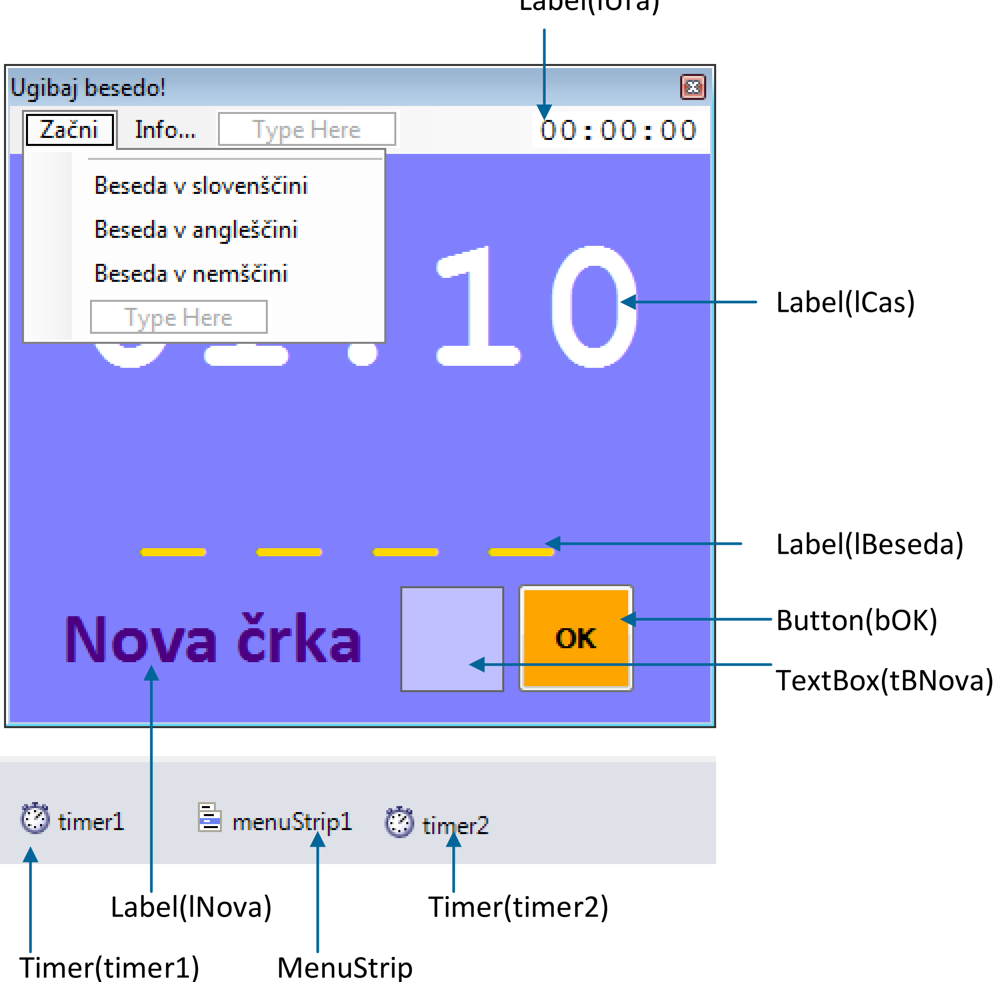
```
//klik na gumb Začetek
private void bZacetek_Click(object sender, EventArgs e)
{
    Start = DateTime.Now;     //začetek merjenja časa
    casTece = true; //v gradniku textBox1 se prikazuje izmerjeni čas
    timer1.Enabled = true;   //poženemo timer
    bVmesni.Visible = true;  //prikažemo gumb za vmesne rezultate
    bKonec.Visible = true;   //prikažemo gumb za končni rezultat
    bZacetek.Visible = false;//skrijemo gumb za začetek
    //na začetku vedno pobrišemo prejšnje rezultate
    listBox1.Items.Clear();
}
//klik na gumb Konec
private void bKonec_Click(object sender, EventArgs e)
{
    End = DateTime.Now; //shranimo trenutni čas
    timer1.Enabled = false;    //zaustavimo timer (merjenje časa)
    //Končni čas zapišemo tudi v ListBox1
```

```
    listBox1.Items.Add("Končni čas: " + textBox1.Text);
    bVmesni.Visible = false;  //skrijemo gumb za vmesne rezultate
    bKonec.Visible = false;   //skrijemo gumb za končni rezultat
    bZacetek.Visible = true;  //prikažemo gumb za začetek
}
//klik na gumb Vmesni čas
private void bVmesni_Click(object sender, EventArgs e)
{
    if(casTece==true)
    {
        casTece = false; //zaustavimo prikazovanje časa v gradniku textBox1
        bVmesni.Text = "Nadaljuj";
        //v listBox1 zapišemo vmesni rezultat
        listBox1.Items.Add(stevec + ".)  " + textBox1.Text);
        stevec++;
    }
    else
    {   //nadaljujemo s prikazom časa v gradniku textBox1
        casTece = true;
        bVmesni.Text = "Vmesni čas";
    }
}
```

Ugibaj besedo

Ugibaj besedo je igra, pri kateri si prijatelj (v tem primeru naš računalnik) izbere neko besedo, mi pa moramo v določenem času ugotoviti, katera beseda je to. Da bo zadeva bolj zanimiva, bo naš program trojezičen – torej bomo lahko ugibali slovenske, angleške ali nemške besede. Kateri jezik bo izbran, bo ena od možnosti, ki si jo bo iz menija izbral uporabnik.

Kot vedno, si tudi sedaj najprej pripravimo videz uporabniškega vmesnika. Prikazan je na sliki Slika **72:** Gradniki na obrazcu *fUgibajBesedo*



Slika 72: Gradniki na obrazcu f*UgibajBesedo*

V projektu bomo uporabili dva gradnika *Timer*. Prvi gradnik *Timer* bomo uporabili za iztekajoči se čas, drugega pa za prikaz tekočega časa. V projektu bomo uporabili tudi gradnik *MenuStrip*, s katerim bomo ustvarili preprosti meni. Spomnimo se, da smo s tem gradnikom delali v razdelku *Delo z enostavnimi in sestavljenimi meniji, orodjarna*.

Obrazec poimenujmo *fUgibajBesedo* in nanj postavimo glavni meni (gradnik MenuStrip), ter dva gradnika tipa *Timer*. Gradniku *timer1* priredimo dogodek, ki bo vsako sekundo odšteval čas, prikazan na osrednji oznaki na obrazcu, gradniku *timer2* pa dogodek, ki bo na oznaki v zgornjem desnem robu obrazca vsako sekundo osveževal trenutni čas.

Gradnik	Lastnost	Nastavitev	Opis
Form1	FormBorderStyle	FixedToolWindow	Uporabnik velikosti okna ne more spreminjati
	Name	fUgibajBesedo	Ime obrazca
	StartPosition	CenterScreen	Obrazec se bo odprl na sredini

			zaslona
lCas	Font	Courier New, Bold, 72	Pisava
	BackColor	128:128;255	Barva ozadja
	ForeColor	White	Bela barva znakov
	Text	01:10	Začetni čas
lBeseda	Font	Courier New, Bold, 36	Pisava
	Text	- - - -	Začetni tekst
tBNova	BackColor	192;192;255	
	BorderStyle	FixedSingle	Enojni okvir okoli polja z besedilom
	CharacterCasing	Upper	Avtomatska pretvorba vnesenih črk v velike črke
	Font	Calibri, Bold, 28	Pisava
	ForeColor	Navy	
	MaxLength	1	Možen je vnos le enega znaka
	TextAlign	Center	Znak bo poravnan na sredini kvadratka
timer1	Interval	1000	Časovni interval je 1 sekunda
timer2	Interval	100	Časovni interval je destinka sekunde
lUra	Font	Courier New, 12	Pisava
	Text	00:00:00	Začetni tekst (ura)
	Anchor	Top,Right	Oznaka je pripeta v zgornji desni rob obrazca
	BackColor	ControlLightLight	Barva ozadja

Tabela 22: Lastnosti gradnikov na obrazcu f*UgibajBesedo*.

V konstruktorju tega obrazca zapišimo kodo, s katero obrazec pred odpiranjem tako pomanjšamo, da je na začetku viden le zgornji del. Opcija *Info…* glavnega menija izpiše sporočilno okno z navodili za uporabo programa.

Pogojni operator '?' je edini operator v C#, ki potrebuje tri sestavne dele: pogoj in dva stavka, ločena s podpičjem. (*pogoj* ? vrednost1 : vrednost2) Logika operatorja je naslednja: če je izpolnjen pogoj, ki se nahaja na levi strani operatorja, vrni

vrednost, ki se nahaja na levi strani podpičja (*vrednost1*), sicer pa vrednost na desni strani podpičja (*vrednost12*).

Na začetku iz datoteke *Besede.txt* (ta se mora nahajati v isti mapi kot izvršilna datoteka) preberemo vse besede in jih shranimo v tabelo *besede*. Pri delu z datoteko v program vključimo tudi varovalni blok.

```
public partial class fUgibajBesedo : Form
{
    string[] besede; //tabela, v katero bomo shranili besede iz datoteke
    public fUgibajBesedo()
    {
        InitializeComponent();
        this.Height = 200;//začetna višina obrazca
        //zaradi dela z datoteko uporabimo varovalni blok
        try
        {
           //vse besede iz datoteke Besede.txt shranimo v tabelo besede
           string[] besede = File.ReadAllLines("Besede.txt");
        }
        catch { MessageBox.Show("Napaka pri delu z datoteko!"); }
    }
    string pomozna="",beseda;
    int sekunde; //spremenljivka, s katero bomo merili čas
    int steviloKorakov = 0;
    private void timer1_Tick(object sender, EventArgs e)
    {
        sekunde--;
        string TimeInString = "";
        int min = sekunde / 60; //celoštevilsko deljenje, dobimo minute
        int sek = sekunde % 60; //preostanek so sekunde
        if (sekunde >= 0)
        {
            /*če je število minut manjše od 10, nizu dodamo še ničlo,
             sicer min le spremenim v niz*/
            TimeInString = ((min < 10) ? "0" + min.ToString() :
min.ToString());
            /*če je število sekund manjše od 10, pred niz sek dodamo še
             ničlo, sicer niz sek le spremenim v nz*/
            TimeInString += ":"+((sek<10)?"0"+sek.ToString():sek.ToString());
            //dobljeni niz, ki predstavlja tekoči čas prikažem na oznaki
            lCas.Text = TimeInString;
        }
        else
        {
            timer1.Stop();//ko se čas izteče se timer zaustavi
            MessageBox.Show("Žal je čas potekel!\n\nIskana beseda je bila
"+beseda+"\n\nOb kliku na OK se bo program zaprl!");
            Close();
        }
    }
    private void timer2_Tick(object sender, EventArgs e)
```

```
    {
        //prikaz tekoče ure v zgornjem desnem delu obrazca
        lUra.Text = DateTime.Now.ToLongTimeString();
    }
    private void infoToolStripMenuItem_Click(object sender, EventArgs e)
    {
        MessageBox.Show("V meniju klikni 'Začni' in izberi jezik, v katerem
boš ugibal besedo.\n\nProgram izbere naključno besedo iz datoteke Besede.txt,
ki se nahaja v isti mapi, kot program.\n\nČas, ki ga imaš na voljo je odvisen
od izbire jezika!");
    }
}
```

Ko uporabnik izbere eno od treh ponujenih opcij za ugibanje besede, program iz tabele *besede* izbere naključno vrstico in iz nje besedo v ustreznem jeziku. V vsaki vrstici te datoteke je namreč beseda napisana v treh jezikih, vmes pa so znaki podpičje. Najprej je beseda v slovenščini, nato v angleščini in na koncu še v nemščini. Glede na število črk v tej besedi program izračuna velikost obrazca in na sredino obrazca izpiše toliko črtic, kot je dolžina besede.

```
private void Ugibaj(object sender, EventArgs e)
{
    Random naklj = new Random();
    //iz tabele vzamemo naključno celico
    string celica = besede[naklj.Next(0, besede.Length)];
    /*v nizu celica so 3 besede (v SLO, ANG in NEM), ločene s
      podpičjem. Z metodo Split jih shranimo v tabelo tabBesed*/
    string[] tabBesed = celica.Split(';');
    if (sender == besedaVSlovenščiniToolStripMenuItem)
    {
        beseda = tabBesed[0].ToUpper();//vse črke naj bodo velike
        sekunde = 70;//na voljo imamo 70 sekund
        this.Text = "Ugibaj besedo v slovenščini!";
    }
    else if (sender == besedaVAngleščiniToolStripMenuItem)
    {
        beseda = tabBesed[1].ToUpper();//vse črke naj bodo velike
        sekunde = 80;//na voljo imamo 80 sekund
        this.Text = "Ugibaj besedo v angleščini!";
    }
    else
    {
        beseda = tabBesed[2].ToUpper();//vse črke naj bodo velike
        sekunde = 90;//na voljo imamo 90 sekund
        this.Text = "Ugibaj besedo v nemščini!";
    }
    pomozna = "";
    /*spremenljivka pomozna naj na začetku vsebuje toliko
     presledkov, kot je dolžina niza beseda*/
    for (int i = 0; i < beseda.Length; i++)
        pomozna += " ";
    Prikazi();//klic metode, ki pokaže že ugotovljene znake
```

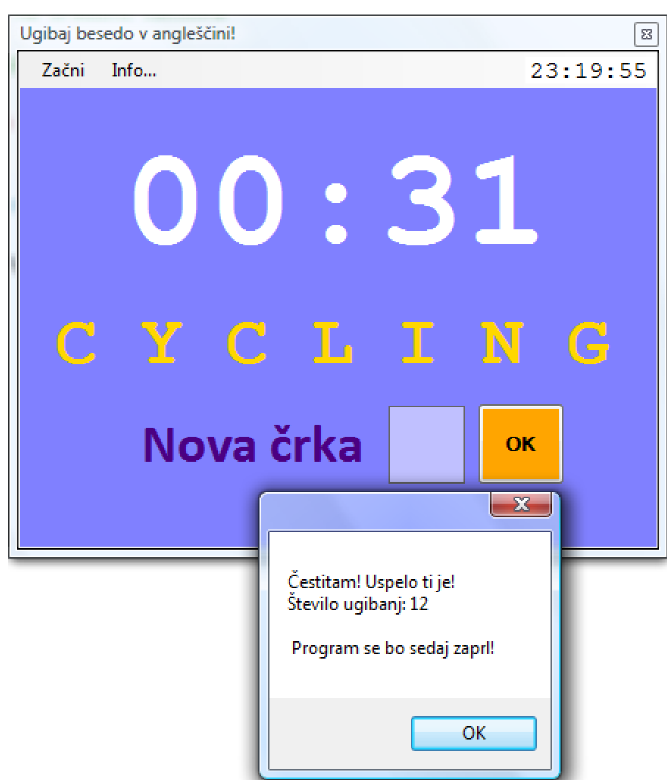
```
    /*gradnike postavimo na sredino okna, ki ne sme biti ožje od 350 px*/
    if (lBeseda.Width < 350)
        this.Width = 350;
    else  //obrazec je za 10 px širši kot je širina besede
        this.Width = lBeseda.Width + 10;
    /*ker je dolžina beseda naključna, moramo vse gradnike postaviti
      na sredino obrazca*/
    lCas.Left = (this.Width - 337) / 2;
    lBeseda.Left = 10 + (this.Width - lBeseda.Width) / 2;
    lNova.Left = (this.Width-(lNova.Width+tBNova.Width + bOK.Width + 5)) / 2;
    tBNova.Left = lNova.Left + lNova.Width + 5;
    bOK.Left = lNova.Left + lNova.Width + bOK.Width + 5;
    tBNova.Focus(); //gradnik za vnos črke postane aktiven
    this.Height = 360; //višina obrazca
    this.CenterToScreen();//okno naj bo na sredini zaslona
    timer1.Start(); //začnemo meriti (odštevati) čas
}
```

Pod črticami je na sredini obrazca vnosno polje za vnos črke. Možen je vnos le enega znaka, to pa je lahko le črka angleške abecede (dogodek *KeyPress*). Na obrazcu je še gumb za potrditev vnosa. Odzivna metoda tega gumba je namenjena preverjanju pravilnosti vnesene črke in prikazu doslej uganjenih črk skrite besede. V ta namen bomo napisali svojo metodo z imenom *Prikazi*.

```
//odzivna metoda za kontrolo vnesenih znakov
private void tBNova_KeyPress(object sender, KeyPressEventArgs e)
{
    //dovoljene tipke so le tipke z znaki
    if (!(e.KeyChar >= 'A' || e.KeyChar <= 'Z'))
        e.Handled = true;
    else bOK.Focus();//gradnik za vnos črke ostane aktiven
}
//odzivna metoda dogodka Click gumba za potrditev vnesenega znaka
private void bOK_Click(object sender, EventArgs e)
{
    if (tBNova.Text.Length > 0)//preverimo, če je uporabnik vnesel črko
    {
        steviloKorakov++;
        /*vnos shranimo v spremenljivko tipa char(znak)*/
        char znak = Convert.ToChar(tBNova.Text);
        string zac = "";//začasna spremenljivka
        /*preverimo, če se ta črka nahaja v iskani besedi: če se, jo
        (jih) umestim na pravilno mesto*/
        for (int i = 0; i < beseda.Length; i++)
        {
            if (beseda[i] == znak)
                zac = zac + znak;
            else
                zac = zac + pomozna[i];
        }
        /*spremenljivka pomozna vsebuje doslej pravilno ugotovljene
        znake, ostali znaki so presledki*/
        pomozna=zac;
```

```
        Prikazi();//klic metode, ki pokaže že ugotovljene znake
        tBNova.Clear(); //pobrišem črko
        tBNova.Focus(); //aktiven gradnik je gumb OK
        if (pomozna==beseda)
        {
            timer1.Enabled = false;//zaustavim čas odštevanja
            MessageBox.Show("Čestitam! Uspelo ti je!\nŠtevilo ugibanj:
"+steviloKorakov+"\n\n Program se bo sedaj zaprl!");
            Close();
        }
    }
}
private void Prikazi()//metoda, ki prikaže že ugotovljene znake
{
    string zac = "";//začasen string
    for (int i = 0; i < beseda.Length; i++)
    {
        if (pomozna[i]!=' ')//če je dosedanji znak presledek
            zac=zac+beseda[i]+" "; //dodam ta znak in še presledek

        else //sicer pa dodam podčrtaj in presledek
            zac=zac+"_"+" ";
    }
    lBeseda.Text = zac;//niz zac prikažem na oznaki
}
```

Še prikaz delujočega projekta



Slika 73: Ugibanje besede.

Program bi lahko nadgradili tudi tako, da bi uporabniku omogočili, da izbira datoteko, kjer so napisane besede. Prav tako pa bi mu lahko dodali še enostaven urejevalnik besed v datoteki s katerim bi kar programsko dodajali (in brisali) besede. Z doslej naučenim to zagotovo ne bo predstavljajo prehudega izziva.

Dinamično ustvarjanje okenskih gradnikov in njihovih dogodkov

Dinamično ustvarjanje okenskih gradnikov

Okenske objekte, ki smo jih uporabljali v naših aplikacijah, smo doslej vselej ustvarjali statično, to je v fazi načrtovanja. V oknu *ToolBox* smo izbrali določen gradnik, ga postavili na določeno mesto na obrazcu in s tem ustvarjali nove okenske objekte. Pri pisanju aplikacij pa se pogosto zgodi, da ne vemo v naprej, koliko gradnikov (gumbov, oznak, vnosnih polj…) bomo potrebovali, ali pa bi radi nek okenski objekt ustvarili šele v fazi izvajanja programa – dinamično. Gradniki, ki se nahajajo na paletah v oknu *ToolBox*, so v bistvu razredi, zato lahko iz njih tvorimo objekte kadarkoli. Vemo, da nov *primerek* (nov objekt) nekega razreda ustvarimo s pomočjo rezervirane besede *new* (to smo spoznali že pri učenju osnov jezika *C#*). Na enak način lahko tvorimo tudi nove objekte tipa *Button*, *Label*, *TextBox*, *ComboBox*…

```
Button gumb = new Button();          //nov objekt tipa Button
Label mojaLabela = new Label();      //nov objekt tipa Label – oznaka
TextBox sporocilo = new TextBox();  //Nov objekt tipa TextBox
ComboBox cBImena = new ComboBox();  //Nov objekt tipa ComboBox
```

S stališča programerja je uporaba besedice *new* pogosto že kar avtomatična – omogoči pač izdelavo novega objekta. Vendar pa je ustvarjanje objekta v resnici dvofazni proces. Operacija *new* mora najprej *alocirati* (zasesti) nekaj prostega pomnilnika na kopici – na ta del ustvarjanja novega objekta nimamo prav nobenega vpliva. Drugi del operacije *new* pa je v tem, da mora zasedeni pomnilnik pretvoriti v objekt – mora *inicializirati* objekt. To drugo fazo operacije *new* pa seveda lahko kontroliramo z uporabo konstruktorja. V zgornjih primerih smo nove objekte izdelali s pomočjo privzetih konstruktorjev, ki nimajo parametrov. Ko je nov objekt ustvarjen, lahko do njegovih članov (polj, metod, ..) dostopamo z uporabo operatorja pika, npr.:

```
sporocilo.Text = "Tekstovno sporočilo!";
gumb.BackColor = Color.Red;
mojaLabela.Visible = false;
```

Za vsak vizuelni objekt, ki ga ustvarimo dinamično, pa moramo še določiti, na kateri gradnik ga želimo postaviti in kakšen je njegov položaj na tem gradniku. Če je obrazec še prazen, in postavljamo nanj dinamično ustvarjen gradnik, to storimo takole

```
//nov objekt tipa Button – ime objekta je gumb
Button gumb = new Button();
gumb.Parent = this; //gumb postavimo na obrazec
gumb.Top = 200; //oddaljenost gumba od vrha obrazca
gumb.Left = 200; //oddaljenost gumba od levega roba obrazca
gumb.BringToFront();//gumb postavimo v ospredje
```

Ključni stavek pri tem je

```
gumb.Parent = this;
```

s katerim določimo, na katerem gradniku leži naš objekt *gumb*. Ker smo znotraj razreda *Form1*, nam *this* seveda označuje prav ta obrazec, ki nastaja.

Oglejmo si še primer, ko želimo dinamično izdelan objekt postaviti na nek že obstoječi gradnik na obrazcu. Recimo, da imamo na obrazcu že gradnik *Panel* z imenom *panel1*, nanj pa želimo dinamično postaviti nov objekt tipa *TextBox*.

```
//ustvarimo nov objekt tipa TextBox – ime mu je tB
TextBox tB = new TextBox();
/*objekt gumb postavimo na gradnik Panel (plošča) z imenom panel1: gradnik
panel1 mora seveda že obstajati!!!*/
tB.Parent = panel1; //objekt tB je postavljen na ploščo z imenom panel1
tB.Top = 20; //oddaljenost od vrha plošče
tB.Left = 20; //oddaljenost od levega roba plošče
tB.BringToFront();
```

Če so na gradniku, na katerem ustvarjamo nov objekt, še drugi gradniki, z metodo *BringToFront* poskrbimo, da bo nov objekt postavljen v ospredje (na ta način zagotovo ne bo skrit za kakim drugim gradnikom!).

Dinamično ustvarjanje okenskih dogodkov

Vsi dinamično ustvarjeni okenski gradniki že poznajo dogodke, ki jih pri statično ustvarjenih objektih najdemo v oknu *Properties→Events*. Njihova implementacija je pri statičnih gradnikih enostavna: v fazi načrtovanja projekta v oknu *Properties→Events* dvokliknemo na ustrezni dogodek in razvojno okolje nam ustvari ogrodje odzivne metode, v katero le še zapišemo ustrezno kodo. Pri dinamično ustvarjenih okenskih gradnikih pa moramo to storiti programsko. To storimo v dveh korakih:

- Najprej moramo poimenovati novo odzivno metodo za dogodek dinamično ustvarjenega gradnika in jo uvrstiti v seznam ostalih metod, ki obdelujejo dogodke. Tem metodam pravimo tudi obdelovalci dogodkov - *event handlers*.

  ```
  /*dinamično ustvarjenemu gradniku mojGumb določimo ime odzivne metode
  dogodka Click: odzivno metodo poimenujmo mojGumb_Click*/
  mojGumb.Click += new EventHandler(mojGumb_Click);
  ```

Popolnoma isto se zgodi tudi pri ustvarjanju odzivnih metod gradnikom, ki jih na obrazec postavljamo pri gradnji projekta. A v tem primeru je razvojno okolje tisto, ki poskrbi za dodajanje metode v seznam, kar se odrazi v datoteki *.designer.cs*

- Odzivno metodo *mojGumb_Click* moramo nato še napisati. Tako kot vse odzivne metode, mora tudi ta vsebovati dva parametra: parameter *sender* (pošiljatelj) in parameter tipa *EventArgs* (parameter s pomočjo katerega lahko pridemo do dodatnih informacij o dogodku).

```
public void mojGumb_Click(object sender, EventArgs e)
{
    //stavki, ki naj se izvedejo ob kliku na dinamično ustvarjen gumb
    MessageBox.Show("Ups! KLiknil si me!");
}
```

Pri dinamičnem ustvarjanju okenskih dogodkov pa nam v veliki meri priskoči na pomoč tudi razvojno okolje. Ko namreč začnemo pisati najavo novega dogodka, se v okvirčku pokaže zelo koristna pomoč (opozorilo: zapis *mojGumb.Click+=…* je krajša oblika zapisa *mojGumb.Click=mojGumb.Click + …)*:

```
mojGumb.Click+=
        new EventHandler(mojGumb_Click);    (Press TAB to insert)
```

Slika 74: Pomoč pri dinamičnem ustvarjanju okenskega dogodka.

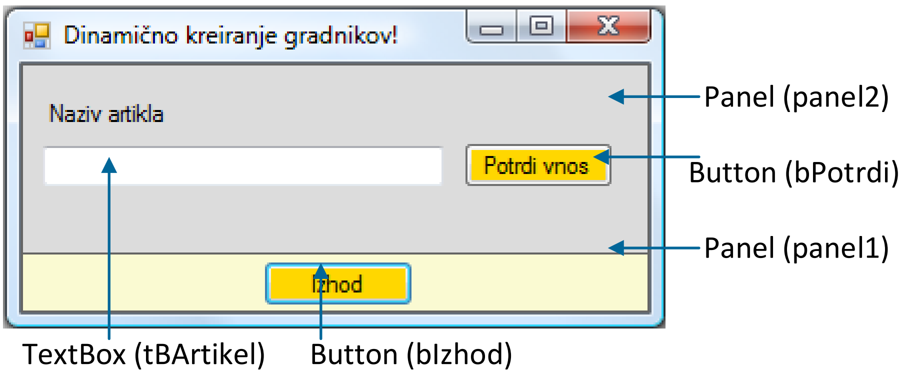
Razvojno okolje ponudi možnost, da najavo novega dogodka zaključimo s pritiskom na tipko tabulator (*Tab*). In še več: če tipko *Tab* pritisnemo, se najava dogodka zapiše do konca (metoda dobi privzeto ime, v našem primeru *mojGumb_Click*), razvojno okolje pa nam ponudi tudi možnost zapisa ogrodja metode, ki jo proži ta dogodek:

```
mojGumb.Click+=new EventHandler(mojGumb_Click);
        Press TAB to generate handler 'mojGumb_Click' in this class
```

Slika 75: Razvojno okolje nam ponudi možnost zapisa ogrodja odzivnega dogodka.

Če torej tipko *Tab* pritisnemo še enkrat, nam razvojno okolje zapiše glavo in ogrodje odzivne metode, mi pa moramo poskrbeti le še za njeno vsebino.

Za vajo ustvarimo obrazec, na katerem bomo vse gradnike (dve plošči, oznako, vnosno polje in dva gumba), ter njihove lastnosti ustvarili dinamično. Dinamično ustvarimo tudi tri dogodke: dogodek *Load* obrazca in dva dogodka *Click* gumbov, ki sta na obrazcu. Končna oblika obrazca naj bo taka:



Slika 76: Dinamično ustvarjeni okenski gradniki na obrazcu.

Vse gradnike bomo ustvraili v metodi *Load*, torej preden se obrazec sploh prikaže. Tako bomo takoj videli vse, tudi dinamično ustvarjene gradnike. Ker v sami kodi ni nič novega, jo takoj navedimo v celoti.

```
public Form1()
{
    InitializeComponent();

    /*Ustvarjanje odzivne metode dogodka Load obrazca Form1 - ime metode bo
      Form1_Load*/
    this.Load += new EventHandler(Form1_Load);
    this.Text = "Dinamično ustvarjanje gradnikov!"; //napis na obrazcu
    this.Width = 330;
    this.Height = 160;
}

private void Form1_Load(object sender, EventArgs e)
{
    //Prva plošča
    Panel panel1 = new Panel();
    panel1.Parent = this; //plošča je postavljena na obrazec
    panel1.Dock = DockStyle.Bottom;  //plošča je prilepljena na dno obrazca
    panel1.Height = 30; //višina lošče
    panel1.BackColor = Color.LightGoldenrodYellow; //barva ozadja
    //plošča ima enojni okvir
    panel1.BorderStyle = BorderStyle.FixedSingle;

    //Druga plošča
    Panel panel2 = new Panel();
    panel2.Parent = this;    //plošča je postavljena na obrazec
    //plošča je raztegnjena čez preostali del obrazca
    panel2.Dock = DockStyle.Fill;
    panel2.BackColor = Color.Gainsboro;  //barva ozadja
    //plošča ima enojni okvir
    panel2.BorderStyle = BorderStyle.FixedSingle;
    //Oznaka
    Label mojaLabela = new Label();
    mojaLabela.Parent = panel2;
```

```
    mojaLabela.Text = "Naziv artikla"; //Napis na oznaki
    mojaLabela.Top = 17;  //oddaljenost od zgornjega roba obrazca
    mojaLabela.Left = 10; //oddaljenost od levega roba obrazca
    //Vnosno polje
    TextBox tBArtikel = new TextBox();
    tBArtikel.Parent = panel2;//TextBox  je postavljen na ploščo panel2
    tBArtikel.Width = 200; //širina vnosnega polja
    tBArtikel.Top = 40;
    tBArtikel.Left = 10;
    //Prvi gumb je ob vnosnem polju
    Button bPotrdi = new Button();

    bPotrdi.Parent = panel2; //Gumb  je postavljen na ploščo panel2
    bPotrdi.Top = 38;
    bPotrdi.Left = 220;
    bPotrdi.Text = "Potrdi vnos";
    bPotrdi.BackColor = Color.Gold;  //barva ozadja
    /*Napoved dogodka Click gumba bPotrdi - ime odzivne metode bo
      bPotrdi_Click*/
    bPotrdi.Click += new EventHandler(bPotrdi_Click);
    Button bIzhod = new Button(); //Drugi gumb je na spodnji plošči
    bIzhod.Parent = panel1; //Gumb je postavljen na ploščo panel1
    bIzhod.Top = 3;
    bIzhod.Left = 120;
    bIzhod.Text = "Izhod";
    bIzhod.BackColor = Color.Gold;  //barva ozadja
    //Napoved dogodka Click gumba bIzhod - ime odzivne metode bo bIzhod_Click
    bIzhod.Click += new EventHandler(bIzhod_Click);
}

public void bPotrdi_Click(object sender, EventArgs ee)
{
    /*stavki, ki se izvedejo ob kliku na gumb Potrdi
      npr. zapis artikla v datoteko ...*/
}

public void bIzhod_Click(object sender, EventArgs ee)
{
    if (MessageBox.Show("Zaključek programa?", "OPOZORILO!",
        MessageBoxButtons.OKCancel) == DialogResult.OK)
        Close();
}
```

Povzetek

V poglavju so zapisane le osnove izdelave okenskih aplikacij. V primerih smo uporabili večino standardnih gradnikov, spoznali njihove osnovne lastnosti in dogodke. Vemo pa tudi to, kako

ustvarimo nove vizuelne objekte in njihove odzivne metode. Pri gradnji aplikacij pa lahko uporabimo tudi številne brezplačne gradnike, ki so dostopni preko spleta in s katerimi bodo naši projekti vizuelno privlačnejši in sodobnejši. Ponudnikov takih gradnikov je veliko npr. https://www.devexpress.com/Products/Free/NetOffer/#winforms

Obdelali smo tudi enega najkompleksnejših gradnikov *DataGridView*, ki ga bomo potrebovali tudi v zadnjem delu učbenika, pri prikazovanju vsebine tabele iz baze podatkov. Naučili smo se tudi uporabljati že izdelana pogovorna okna, ter spoznali dogodke tipkovnice, miške in ure. Na koncu poglavja smo še spoznali, da lahko objekte gradnikov okenskih aplikacij ustvarjamo tudi dinamično, kar razvijalcem projektov omogoča izdelavo bolj kompleksnih rešitev z manj pisanja kode.

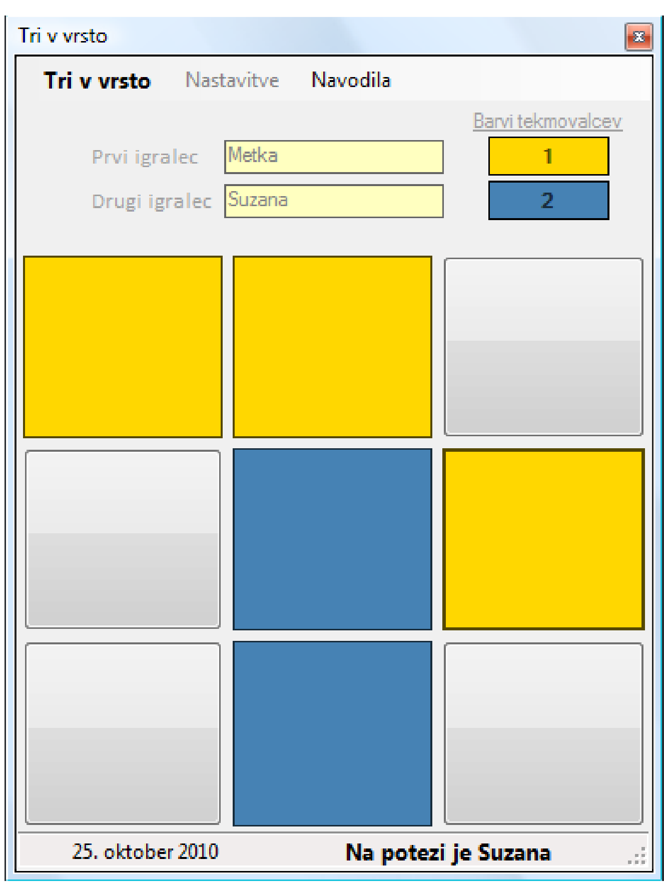
Ker so vsi gradniki primerki objektov iz pripravljenih razredov, zapisani primeri omogočajo, da bo objektni pristop k programiranju postal bolj domač. Znanje pa bo potrebno utrditi tako, da boste napisali kar največ uporabnih programov.

Pridobljeno znanje bomo nadgradili z izdelavo večokenskih aplikacij in vpogledom v naslednja ključna pojma objektno orientiranega programiranja – *dedovanje* in *polimorfizem*.

Kot zaključek spoznavanja osnov izdelave okenskih aplikacij pa sedaj rešimo naš uvodni izziv – program za igranje igrice Tri v vrsto. Seveda se bomo tudi tu spoznali s čim novim in uporabili kak nov gradnik. Osnovno znanje, potrebno za naš program, pa smo si doslej že pridobili.

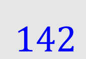
Tri v vrsto

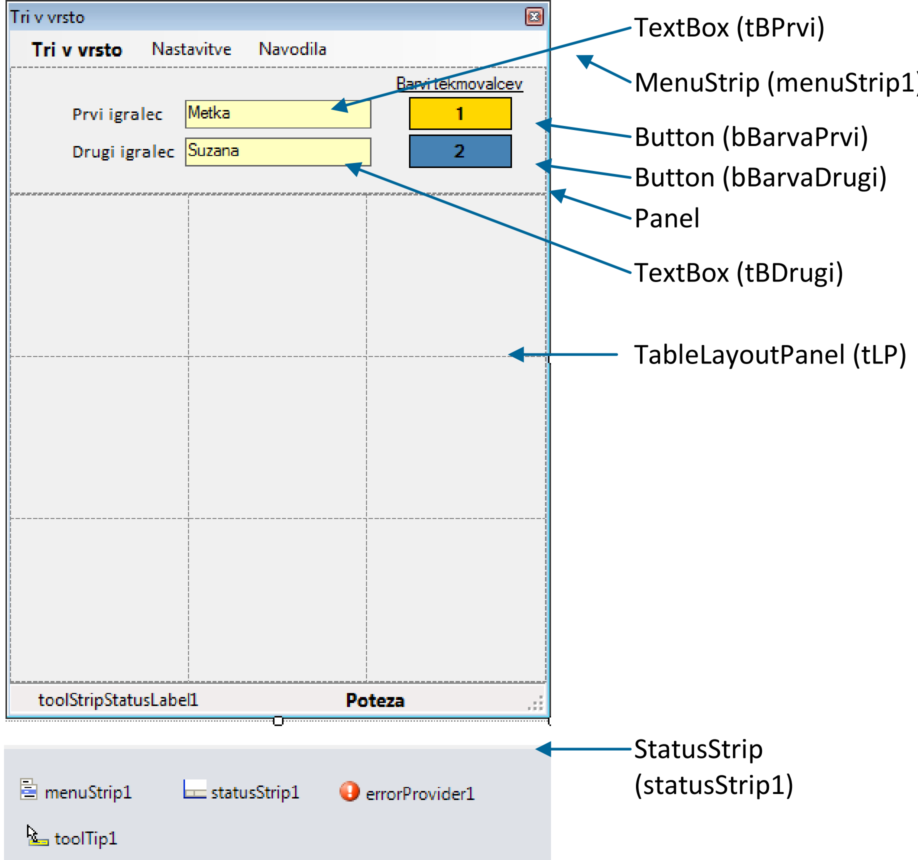
Večino znanja, ki smo ga pridobili pri dosedanjih projektih, uporabimo za rešitev igrice *Tri v vrsto*.



Slika 77: Program *Tri v vrsto* med igro.

Ob pripravi programa bomo spoznali gradnik *TableLayoutPanel*, ki predstavlja nekakšno grafično različico dvodimenzionalne tabele. V tem gradniku bomo dinamično ustvarili devet gumbov (objektov tipa *Button*), jim dinamično določili nekatere lastnosti (velikost, barvo ozadja, stil), dinamično pa bomo ustvarili tudi odzivne metode dogodkov *Click*, *MouseEnter* in *MouseLeave*. Seveda bi lahko rešitev naredili tudi tako, da bi uporabili devet statičnih objektov izpeljanih iz gradnika *Button*, a pregled nad njimi bi bil v tem primeru manj pregleden, koda pa zaradi tega daljša. Za vsakokratno preverjanje zmagovalca in preverjanje, ali je igra končana, bomo napisali tudi nekaj lastnih metod. V rešitvi bomo preverjali tudi uporabnikove vnose, njihova pravilnost pa bo pogoj za začetek igre. Oba igralca bosta s pomočjo dialoga za izbiro barve izbrala svojo igralno barvo.

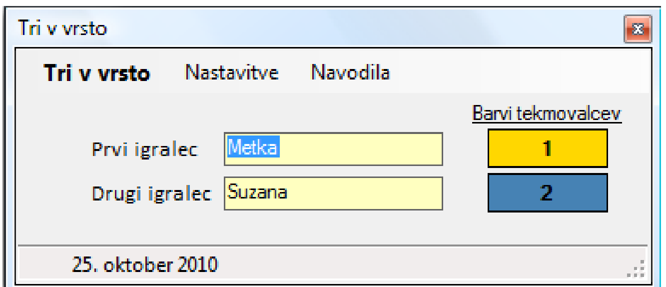




Slika 78: Gradniki obrazca *Tri v vrsto*.

Na prazen obrazec postavimo gradnike v naslednjem zaporedju: *MenuStrip* in *Panel* (*Dock=Top,* nanj pa postavimo tri oznake, dve vnosni polji in dva gumba), *StatusStrip* (na njem sta dva predalčka: prvemu nastavimo lastnost *AutoSize* na *False* in mu določimo širino 150, drugemu pa nastavimo lastnost *Spring* na *True*, lastnost *Text* pa *Poteza*), na koncu pa še gradnik *TableLayoutPanel* (*Dock=Fill*). Gradnik privzeto že vsebuje dva stolpca in dve vrstici: poimenujmo ga *tLP* in mu dodajmo še po en stolpec in eno vrstico (kliknemo na puščico v zgornjem desnem kotu gradnika, nato pa *Add Column* in *Add Row*). Stolpcem in vrsticam nastavimo enake širine in višine (klik na lastnost *Columns* oz. *Rows*, nato pa v oknu, ki se odpre, vsaki postavki posebej določimo lastnost *Percent* na 33.33 procentov!).

V konstruktor obrazca bomo zapisali kodo tako, da bo ob zagonu projekta viden le zgornji del obrazca.



Slika 79: Začetna velikost obrazca *Tri v vrsto*.

Od uporabnika programa namreč najprej zahtevamo, da vnese imeni obeh igralcev, ki imata tudi možnost izbire svoje barve. Izbrana barva (ali pa privzeta barva) je prikazana v okvirčkih s številkama obeh tekmovalcev. V glavnem meniju je tudi opcija *Navodila*, ki v sporočilnem oknu prikaže kratka navodila za samo igro. Na dnu obrazca je statusna vrstica s tekočim datumom. Igro pričnemo s klikom na *Tri v vrsto*.

Med samo igro je v statusni vrstici ves čas navodilo, kateri igralec je trenutno na vrsti (glej napis desno spodaj na sliki Slika **77:** Program *Tri v vrsto* med igro. Ob zmagi enega izmed igralcev (tri enake barve v vrsti, stolpcu ali diagonali), se v sporočilnem oknu izpiše ustrezno obvestilo. Le-to se izpiše tudi v primeru neodločenega rezultata (vsa okna so pobarvana, zmagovalca pa ni!). Ko uporabnik zapre sporočilno okno, se avtomatično prične nova igra.

V konstruktorju obrazca ustvarimo 9 gumbov in jih dodamo v gradnik *TableLayoutPanel*. Za vsakega od gumbov napovemo še odzivne metode za tri dogodke.

```
bool igralec = true;//igro začne prvi igralec
ColorDialog cD = new ColorDialog();//nov objekt za dialog izbire barve
//konstruktor obrazca
public Form1()
{
    InitializeComponent();
    //v okencu gradnika StatusStrip prikažemo datum
    toolStripStatusLabel1.Text = DateTime.Now.ToLongDateString();
    //izdelava gumbov
    for (int i = 0; i < 9; i++)
    {
        Button button = new Button();//nov objekt tipa Button
        button.Text = ""; //gumb je brez napisa
        //gumb dodamo v gradnik TableLayoutPanel
        tLP.Controls.Add(button);
        /*do vsakega gumba v gradniku tLP dostopamo s pomočjo lastnosti
          Controls, oglatega opklepaja in indeksa tega gumba (indeksi se
          začenjajo z 0!)*/
        //gumb zasede celotno celico tabele
        tLP.Controls[i].Dock = DockStyle.Fill;
        //upravljalcu dogodkov dodajmo dogodek Click posameznega gumba
        tLP.Controls[i].Click += new System.EventHandler(OK_Click);
        //upravljalcu dogodkov dodajmo dogodek MouseEnter posameznega gumba
        tLP.Controls[i].MouseEnter += new System.EventHandler(OK_Enter);
        //upravljalcu dogodkov dodajmo dogodek MouseLeave posameznega gumba
        tLP.Controls[i].MouseLeave += new System.EventHandler(OK_Leave);
    }
    this.Height = 160;//Začetna velikost obrazca
    /*vsebina predalčka gradnika StatusStrip, ki označuje kateri igralec je
      na potezi, je na začetku prazna*/
    toolStripStatusLabel2.Text = "";
}
```

Za uspešen začetek igre morata igralca v vnosni polji vpisati svoja imena. Ob kliku na možnost *Tri v vrsto* glavnega menija se igra lahko začne.

```
private void tB_Validating(object sender, CancelEventArgs e)
{
    KontrolaImena();//metoda za preverjanje vnosa imen obeh tekmovalcev
}
private bool KontrolaImena()
{
    bool bStatus = true;
    if (tBPrvi.Text == "")
    {
        /*Če uporabnik ne bo vnesel imena, se bo ob imenu pokazala rdeča
          oznaka s klicajem. Če se bomo z miško postavili na to oznako, se
          pod njim v okvirčku pojavi besedilo, ki ga zapišemo kot drug
          parameter metode SetError */
        errorProvider1.SetError(tBPrvi, "Vnesi ime prvega igralca");
        bStatus = false;
    }
    else errorProvider1.SetError(tBPrvi, "");/*ko je ime vneseno, odstranimo
                                                     sporočilo o napaki*/
    if (tBDrugi.Text == "")
    {
        errorProvider1.SetError(tBDrugi, "Vnesi ime drugega igralca");
        bStatus = false;
    }
    else errorProvider1.SetError(tBDrugi, "");
    return bStatus;//metoda vrne false, če imena tekmovalcev niso vnesena
}
//odzivna metoda za začetek igre
private void triVVrstoToolStripMenuItem_Click(object sender, EventArgs e)
{
    //če sta imeni tekmovalcev vneseni, lahko začnemo z igro
    if (KontrolaImena())
    {
        panel1.Enabled = false;
        this.Height = 500;
        //med samo igro ni možno spreminjati barve posameznega tekmovalca
        nastavitveToolStripMenuItem.Enabled = false;
        Zacetek();
    }
}
//lastna metoda za določanje začetnega stanja
private void Zacetek()
{
    igralec = true;//začenja prvi igralec
    toolStripStatusLabel2.Text = "Na potezi je " + tBPrvi.Text;
    //vsem gumbom gradnika TablePanelLayout določimo začetno barvo in stil
    for (int i = 0; i < 9; i++)
    {
        (tLP.Controls[i] as Button).FlatStyle = FlatStyle.System;
        tLP.Controls[i].BackColor = Color.FromArgb(0,0,0,0);
    }
}
```

Igralca lahko določita svoji barvi gumbov:

```
//odzivna metoda za izbiro barve prvega igralca
private void barvaPrvegaIgralcaToolStripMenuItem_Click(object sender,
EventArgs e)
{
    /*če prvi igralec izbere barvo in klikne gumb OK, mu določimo novo
      barvo*/
    if (cD.ShowDialog() == DialogResult.OK)
    {
        if (bBarvaDrugi.BackColor != cD.Color)
            bBarvaPrvi.BackColor = cD.Color;
        else MessageBox.Show("Barvi obeh tekmovalcev ne smeta biti enaki!");
    }
}
//odzivna metoda za izbiro barve drugega igralca
private void barvaDrugegaIgralcaToolStripMenuItem_Click(object sender,
EventArgs e)
{
    */če drugi igralec izbere barvo in klikne gumb OK, mu določimo novo
      Barvo*/
    if (cD.ShowDialog() == DialogResult.OK)
    {
        if (bBarvaPrvi.BackColor != cD.Color)
            bBarvaDrugi.BackColor = cD.Color;
        else MessageBox.Show("Barvi obeh tekmovalcev ne smeta biti enaki!");
    }
}
```

Dodamo še odzivno metodo postavki *Navodila* glavnega menija. V sporočilnem oknu pojasnimo pravila igre.

```
private void navodilaToolStripMenuItem_Click(object sender, EventArgs e)
{
    MessageBox.Show("Vnesi imeni obeh tekmovalcev, v meniju 'Nastavitve'
izberi barvo posameznega tekmovalca, nato pa v meniju klikni 'Tri v
vrsto'!\r\n\r\nBarvo igralca lahko izbereš tudi s klikom na gumb 1 oz
2!","Kratka navodila",0,MessageBoxIcon.Information);
}
```

Odzivne metode za vstop in izstop miškinega kazalca v območje gumba, ter metoda za klik na gumb, so enake za vse gumbe v gradniku *TableLayoutPanel*. Pri tem bmo uporabili parameter *sender* odzivnih metod, ki smo ga spoznali v poglavju *Parametri odzivnih metod*.

```
//dogodek se izvede, ko z miško vstopimo v območje nekega gumba
public void OK_Enter(object sender, EventArgs ee)
{
    //ko z miško vstopimo v območje gradnik, se oblika gumba spremeni
    (sender as Button).FlatStyle=FlatStyle.Popup;
}
//dogodek se izvede, ko z miško zapustimo območje nekega gumba
```

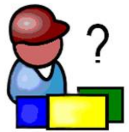
```
public void OK_Leave(object sender, EventArgs ee)
{
    /*če z miško zapustimo gumb, ki še ni pobarvan, bo njegova oblika taka,
    kot pred vstopom*/
    if ((sender as Button).BackColor==Color.FromArgb(0,0,0,0))
        (sender as Button).FlatStyle=FlatStyle.System;
}

public void OK_Click(object sender, EventArgs ee)
{
    if ((sender as Button).BackColor == Color.FromArgb(0, 0, 0, 0))
    {
        Color barva;
        if (igralec)//če gre za prvega igralca
        {
            /*barva za prvega igralca je določena z ozadjem gumba
              bBarvaPrvi*/
            barva = bBarvaPrvi.BackColor;
            (sender as Button).BackColor = barva;
            /*v predalček gradnika StatusStrip zapišemo, da je na potezi
              drugi igralec*/
            toolStripStatusLabel2.Text = "Na potezi je " + tBDrugi.Text;
        }
        else
        {
            /*barva za drugega igralca je določena z ozadjem gumba
              bBarvaDrugi*/
            barva = bBarvaDrugi.BackColor;
            (sender as Button).BackColor = barva;
            /*v predalček gradnika StatusStrip zapišemo, da je na potezi
              prvi igralec*/
            toolStripStatusLabel2.Text = "Na potezi je " + tBPrvi.Text;
        }
        //preverimo, če je zmagal igralec, ki je bil pravkar na potezi
        if (PreveriRezultat(barva))
        {
            /*izpraznimo vsebino predalčka gradnika StatusStrip, ki
             označuje kateri igralec je na potezi*/
            toolStripStatusLabel2.Text = "";
            //preverimo, za katerega igralca gre in izpišemo zmagovalca
            if (igralec)preverimo, za katerega igralca gre
                MessageBox.Show("KONEC - Zmagovalec je " + tBPrvi.Text);
            else MessageBox.Show("KONEC - Zmagovalec je " + tBDrugi.Text);
            //določimo začetno velikost obrazca (skrijemo spodnji del)
            this.Height = 160;//Začetna velikost obrazca
            //omogočimo dostop do gradnikov na plošči panel1
            panel1.Enabled = true;
            /*izpraznimo vsebino predalčka gradnika StatusStrip, ki
              označuje kateri igralec je na potezi*/
            toolStripStatusLabel2.Text = " ";
            //omogočimo izbiro barv obeh igralcev
            nastavitveToolStripMenuItem.Enabled = true;
        }
```

```
        /*preverimo še, če so vsa polja že polna, kar pomeni, da je rezultat
          neodločen*/
        else if (Neodloceno())
        {
            /*izpraznimo vsebino predalčka gradnika StatusStrip, ki
              označuje kateri igralec je na potezi*/
            toolStripStatusLabel2.Text = "";
            MessageBox.Show("Rezultat je neodločen!", "Igra je končana", 0,
MessageBoxIcon.Information);
            this.Height = 160;//Začetna velikost obrazca
            //omogočimo dostop do gradnikov na plošči panel1
            panel1.Enabled = true;
            //omogočimo izbiro barv obeh igralcev
            nastavitveToolStripMenuItem.Enabled = true;
        }
        else igralec = !igralec;//zamenjamo igralca, ki je na vrsti
    }
}
```

Po vsakem kliku gumba smo glede na igralca pobarvali kliknjen gumb in takoj preverili stanje igre. Najprej smo s sklicem logične metode *PreveriRezultat* preverili, če je igra končana in imamo zmagovalca. V kolikor zmagovalca ni, vsi gumbi pa so poklikani, s pomočjo metode *Neodloceno* preverimo morebiten neodločen rezultat. Tule sta obe metodi.

```
//metoda vrne True, če imamo zmagovalca
private bool PreveriRezultat(Color barva)
{
    /*Preverimo vse možne kombinacije za tri v vrsto. Metoda vrne true, če
      imamo zmagovalca*/
    if ((tLP.Controls[0].BackColor==barva &&
tLP.Controls[1].BackColor==barva&&tLP.Controls[2].BackColor==barva)||
        (tLP.Controls[3].BackColor == barva && tLP.Controls[4].BackColor ==
barva && tLP.Controls[5].BackColor == barva)||
        (tLP.Controls[6].BackColor == barva && tLP.Controls[7].BackColor ==
barva && tLP.Controls[8].BackColor == barva)||
        (tLP.Controls[0].BackColor == barva && tLP.Controls[3].BackColor ==
barva && tLP.Controls[6].BackColor == barva)||
        (tLP.Controls[1].BackColor == barva && tLP.Controls[4].BackColor ==
barva && tLP.Controls[7].BackColor == barva)||
        (tLP.Controls[2].BackColor == barva && tLP.Controls[5].BackColor ==
barva && tLP.Controls[8].BackColor == barva)||
        (tLP.Controls[0].BackColor == barva && tLP.Controls[4].BackColor ==
barva && tLP.Controls[8].BackColor == barva)||
        (tLP.Controls[2].BackColor == barva && tLP.Controls[4].BackColor ==
barva && tLP.Controls[6].BackColor == barva))
        return true;
    else return false;
}
/*metoda vrne True, če je igra končana (vsa polja so poklikana) in zmagovalca
ni*/
private bool Neodloceno()
{
    //če so že vsa polja izpolnjena metoda vrne true: rezultat je neodločen
```

```
    bool neodl = true;
    for (int i=0;i<9;i++)
        if (tLP.Controls[i].BackColor==Color.FromArgb(0,0,0,0))
            neodl=false;
    return neodl;
}
```



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 tri razrede: prvega bi poimenovali *Sesalec* drugega, *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 razred *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, lahko dodamo v razred *Konj* oz. v 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 en razred. 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
{
    public int x, y;      // koordinati točke
    public Tocka()    //privzeti konstruktor
    {
        x = 0; y = 0; //ta dva stavka pravzaprav nista potrebna
    }
    public Tocka( int vrednostX, int vrednostY )    // konstruktor
    {
        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;//zaščiteno polje, vidno tudi v izpelj. razredih
    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
```

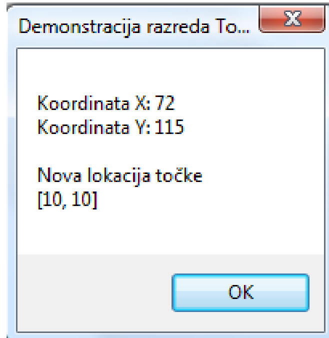
```
    {
        return Math.PI *polmer * polmer;
    }
    // metoda vrne niz, ki predstavi krog
    //še bolje public override Opis(razlaga v nadaljevanju)
    public string Opis()
    {
        return "Središče kroga = [" + x + ", " + y +"], polmer = " +
polmer+"\nPloščina: "+Math.Round(PloscinaKroga(),2);
    }
} // 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.

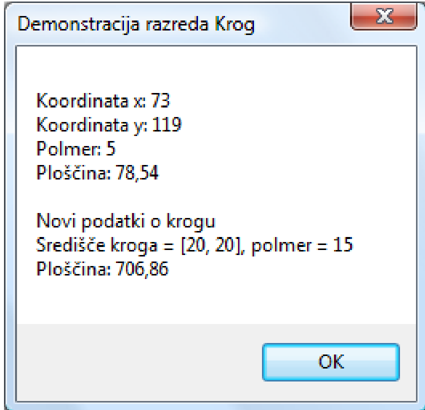
```
Tocka t = new Tocka( 72, 115 );//objekt razreda Tocka
// trenutni vrednosti koordinat zapišemo v niz izpis
string izpis = "Koordinata X: " + t.x + "\nKoordinata Y: " + t.y;
t.x = 10; t.y = 10;   // Določimo novi koordinati
// nizu izpis dodamo še novi vrednosti koordinat
izpis += "\n\nNova lokacija točke\n" + t.Opis();
MessageBox.Show(izpis, "Demonstracija razreda Točka");
```



Slika 80: 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 81: 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*.

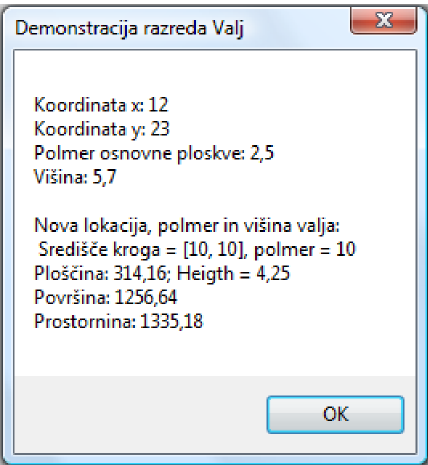
```
public class Valj : Krog
{
    public double visina; // dodatno polje razreda Valj
    public Valj()    // privzeti konstruktor
    {
        visina = 0.0;
    }
    // dodatni konstruktor
    public Valj( int x, int y,double polmer, double h )
        : base(x, y, polmer)// klic konstruktorja razreda 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 82: 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 uporabiti 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.

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()"
```

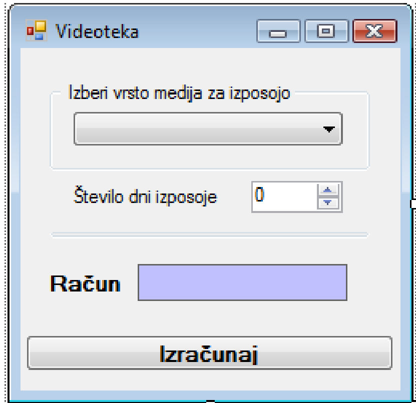
To pa je osnova načela, ki se imenuje p*olimorfizem*. Izvede se metoda razreda, iz katerega je objekt izpeljan.

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, na njem pa *ComboBox* za izbiro medija in gradnik *NumericUpDown* za vnos števila dni izposoje. Temu gradniku pustimo privzeto lastnost *minimum* enako 0, lastnost *maximum* pa nastavimo poljubno, npr. 365 (maksimalno število dni izposoje bo eno leto). Dodajmo še oznako za izpis rezultata in gumb za izračun rezultata.



Slika 83: 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. Imenuje se *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;
```

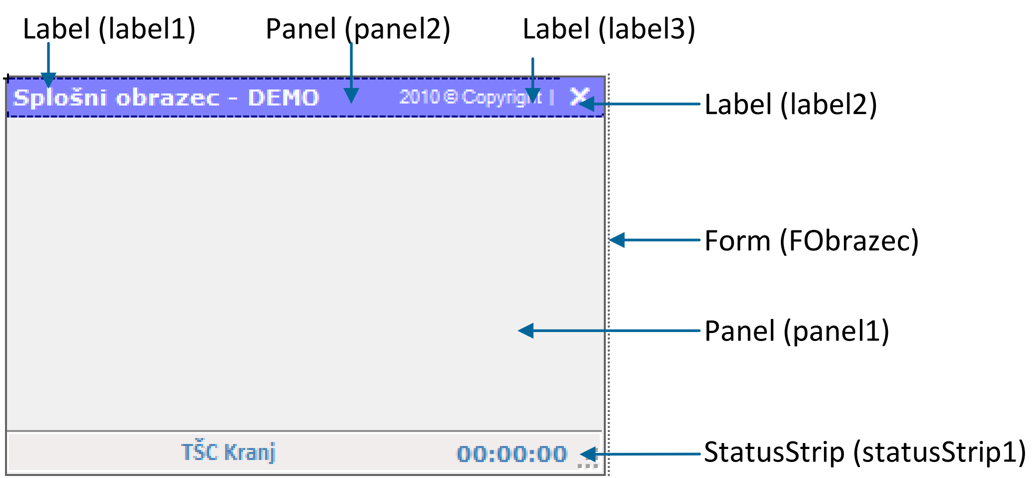
V telo razreda *Form1* datoteke *Form1.cs*, pa smo že doslej pisali odzivne dogodke in metode.

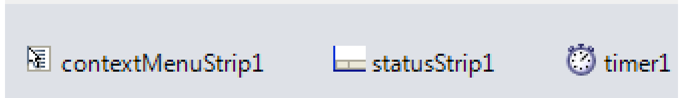
```
public partial class Form1 : Form  //Form1 je izpeljan iz razreda Form
{
    public Form1()//Konstruktor razreda Form1
    {
       InitializeComponent();
    }
    //Odzivni  dogodki  in metode razreda Form1
}
```

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 dedovali 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, v enakem zaporedju jim tudi nastavljamo lastnosti.





Slika 84: 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	Barva 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
contextMenuStrip1	Items • toolStripMenuItem1 ○ Text	Info	Napis na oznaki menija
	Modifiers	Protected	Lebdeči meni bomo lahko na podedovanih obrazcih dopolnjevali

Tabela 23: 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 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();
    }
}
```

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, ki se nahaja v imenskem prostoru Obrazec
public partial class Form1 : Obrazec.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, ne 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 *Inheritance 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;
    }
    public int IdOsebe //Lastnost oz. Property
    {
        get {return idOsebe;}
        set {idOsebe = value;}
    }
    public string Ime //Lastnost oz. Property
    {
        get { return ime;}
        set { ime = value;}
    }
    public string Priimek //Lastnost oz. Property
    {
        get { return Priimek;}
        set { Priimek = value;}
    }
    public string Naslov //Lastnost oz. Property
    {
        get { return naslov; }
        set { naslov = value; }
    }
    public string Kraj //Lastnost oz. Property
    {
        get { return Kraj; }
        set { Kraj = value; }
    }
    public int Posta //Lastnost oz. Property
    {
        get { return posta; }
        set
        {   //veljavna poštna številka je npr. med 1000 in 10000
            if (value > 1000 && value < 100000)
                posta = value;
            else posta = 0;
        }
    }
    //metoda Izpis je virtualna - v izpeljanem razredu jo bomo 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; ;
            }
    }
    public override string Izpis() //prepišemo bazično metodo Izpis
    {
        return base.Izpis()+" "+krvnaSkupina+" "+stDarovanj;
    }
}
```

Pokažimo še, kako lahko nov vizuelni gradnik izpeljemo iz obstoječega. Postopek si oglejmo na primeru razvoja gradnika tipa oznaka (*Label*), 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*, jo poimenujmo *UtripajocaOznaka* in kliknimo *Add*. V oknu *Solution Explorer* se pojavi nov element *UtripajocaOznaka*, v urejevalniku nov zavihek *UtripajocaOznaka.cs*, v njem pa 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*.

Gradniku *Timer* zapišimo še odzivno metodo dogodka *Tick*, obenem pa v konstruktorju novega gradnika zapišimo privzeti font.

Slika 85: Nov gradnik: utripajoča oznaka.

```
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.

```
//lastnost/Property novega gradnika UtripajocaOznaka
public int Interval
{
    get { return timer1.Interval; }
    set { timer1.Interval = value; }
}
```

Dodajmo še lastnost, s pomočjo katere bomo imeli dostop in možnost spreminjanja besedila oznake label1.

```
public string Tekst //Property za dostop/spreminjanje besedila oznake
{
    get { return label1.Text; }
    set { label1.Text = 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 v nadaljevanju storili z gradnikoma *NumberBox* in *Gumb*. Interval utripanja lahko spremenimo kar v oknu *Properties* s pomočjo nove lastnosti *Interval*.

Dodajmo še razreda *NumberBox* in *Gumb*, ki bosta dedovala vizuelna gradnika *TextBox* in *Button*. Razred *NumberBox* je v bistvu *TextBox*, v katerega pa lahko vnašamo le števke in eno samo 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 obdelovalca dogodkov *KeyPressEventHandler* in *CancelEventHandler*. Njuna naloga je kontrola uporabnikovega vnosa v vnosno polje. Preverjamo pravilnost vnosa in dejstvo, ali je uporabnik vrednost sploh vnesel.

Preden začnemo pisati stavke, s katerimi bi ustvarili nov razred, ki bo dedoval nek vizuelni gradnik, pa je potrebno v projekt vključiti ustrezne imenske prostore (*using* stavke). Vendar pa pozor: imenski prostor *System.Windows.Forms* lahko vključimo v knjižnico le v primeru, da smo vanjo že dodali vsaj eno uporabniško kontrolo (*User Control*).

```
using System.Windows.Forms;//imenski prostor za dostop do razreda TextBox
//imenski prostor za dostop do razreda KeyPressEventHandler-validacija vnosa
using System.ComponentModel;
using System.Drawing; //dodana imenski prostor za dostop do razreda Color
```

Ko smo dodali ustrezne imenske prostore, lahko pričnemo z ustvarjanjem razredov, ki dedujejo poljubne gradnike za gradnjo aplikacij, ki jih že poznamo.

```
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);
    }
    /*odzivna metoda dogodka Validating, ki se zgodi, ko se uporabnik
      premakne na drug gradnik*/
    private void NumberBox_Validating(object sender, EventArgs e)
    {
        KontrolaVnosa(); //klic metode
```

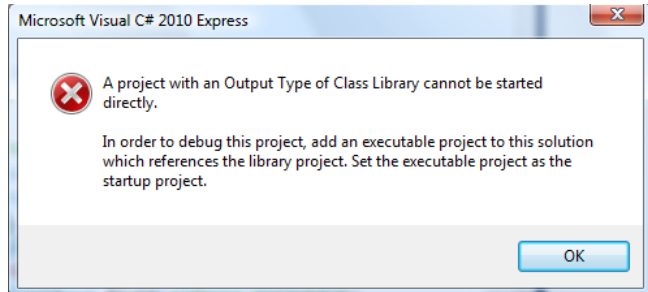
```
    }
    /*metoda vrne true, če uporabnik vnese vsaj en znak, sicer pa generira
      ustrezno sporočilo*/
    private bool KontrolaVnosa()
    {
        if (this.Text == "")
        {
            eP.SetError(this, "Vsebina polja ne sme biti prazna!");
            return false;
        }
        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: vneseni znak se ne pokaže v polju*/
        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);
    }
}
```

Dodajmo še razred Gumb. Ta naj 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 razreda Gumb
    {   //našemu gumbu nastavimo nekaj osnovnih lastnosti
        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 86: 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.

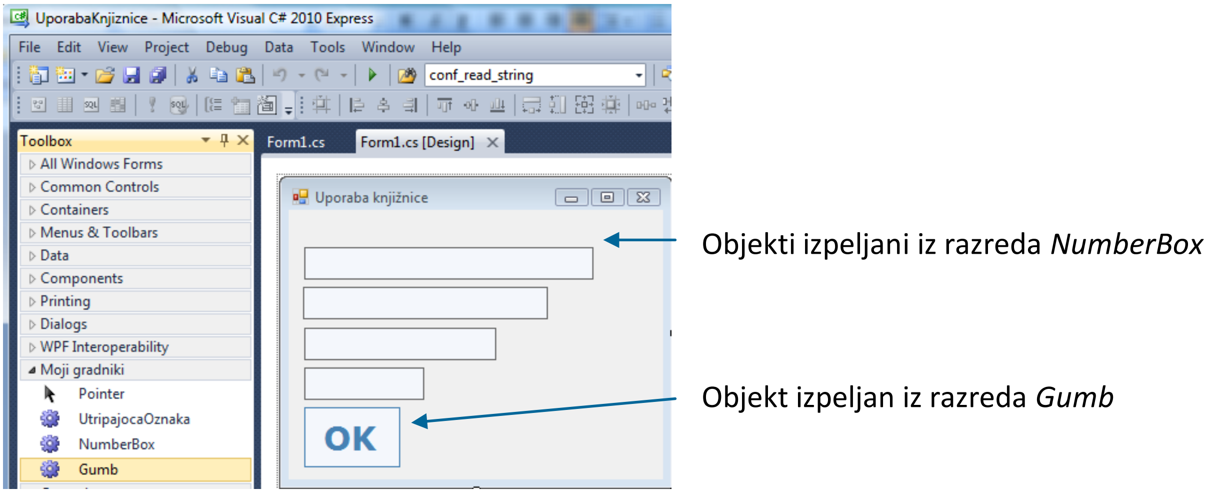
Uporaba lastne knjižnice

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.



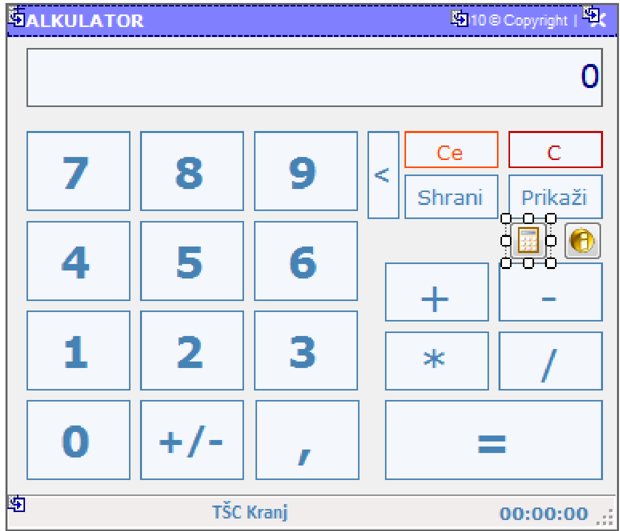
Slika 87: Nova gradnika *NumberBox* in *Gumb* v oknu *Toolbox*.

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*→*in* poiščemo 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 : Obrazec.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 88: Kalkulator.

Na začetku poskrbimo za začetne vrednosti spremenljivk. Prikazovalnik rezultata bo onemogočen ves čas, gumb za izračun rezultata pa le na začetku. Ko se obrazec prikaže prvič, aktiviramo prikazovalnik in žarišče 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; //na začetku onemogočimo gumb za izračun
        numberBox1.Enabled = false; //skrijemo kurzor v vnosnem polju
    }

    /*na začetku mora biti izbran prikazovalnik, žarišče 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 { }
}
```

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 s klikanjem miške. Ob pritisku numerične tipke na tipkovnici, ali pa kliku miške na nek gumb s števko, se bo v prikazovalniku prikazala ustrezna števka. 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 števka 0, jo prej odstranimo
        if (numberBox1.Text == "0" || numberBox1.Text=="-0")
            numberBox1.Clear();
        numberBox1.Text += e.KeyChar;//dodam vtipkano števko
    }
    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 števkami 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 števko
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 števko
```

```
    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
    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 izpeljanih razredih. Razred za abstraktnega proglasimo s pomočjo rezervirane besede *abstract*.

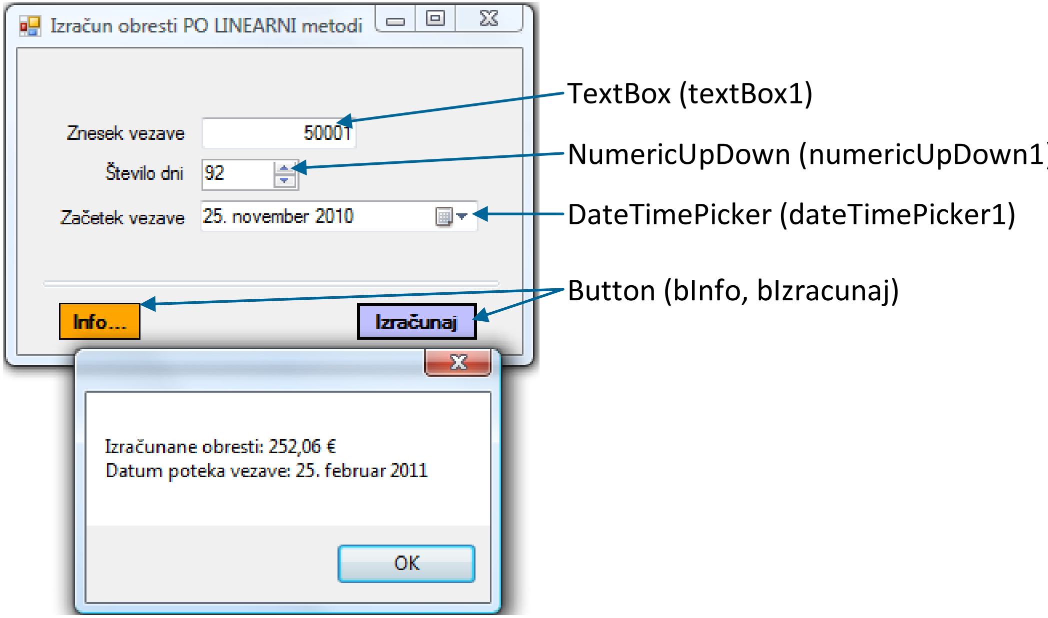
V abstraktnih razredih se pogosto pojavijo tudi *abstraktne metode*. Pred tipom take metode prav tako zapišemo besedico *abstract*. Abstraktne metode imajo samo glavo, ki ji sledi podpičje,

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 89: Obrazec za izračun obresti.

Abstraktni razred naj vsebuje podatka o imenu komitenta in stanju na računu, ter ustrezen konstruktor. Dodajmo mu še objektno metodo *Transakcija*, ki bo skrbela za ažuriranje stanja

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 enakim 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*.

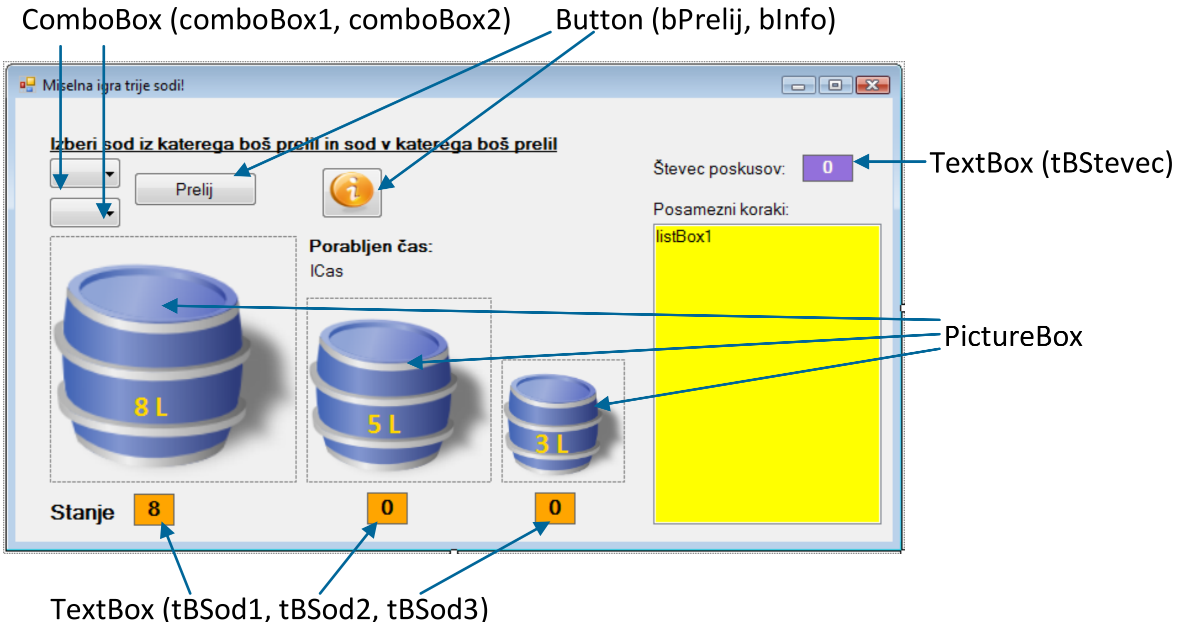
```
//trenutne obresti so konstante
const double obresti1 = 1.90; //obresti za zneske do 10000 EUR
const double obresti2 = 1.95; //obresti za zneske od 10001 do 50000 EUR
const double obresti3 = 2.00; //obresti za zneske nad 50000 EUR

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 v prvih dveh sodih natanko 4 litre tekočine.



Slika 90: Miselna igra trije sodi!

Za potrebe projekta ustvarimo abstraktni razred *Valj*, 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 napišimo 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.

```
//odzivna metoda gumba bPrelij
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 metodi 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.

```
//lastna metoda za prelivanje iz soda v sod
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;
    //preverimo, če je prelivanje končano
    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 deduje abstraktni razred torej nima dostopa do originalne implementacije take metode – v zgornjem primeru torej metoda *Nekaj* razreda *F* **NE** more klicati metode *Nekaj* razreda *Osnovni*.

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 pokažemo, kako izdelamo dodatne obrazce in kako jih odpiramo, pa moramo vedeti, 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 žarišče oz. 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.

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.*

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

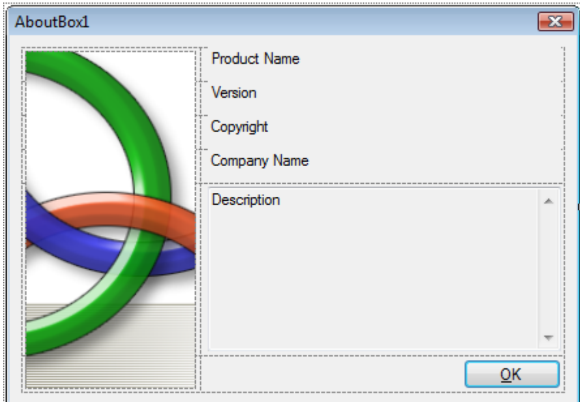
Podobno lahko iz kateregakoli projekta poljuben obrazec tudi izključimo. Postopek je podoben kot pri dodajanju. V oknu *Solution Explorer* obrazec najprej izberemo, kliknemo desni miškin gumb in nato izberemo opcijo *Delete*. Prikaže se pogovorno okno, v katerem lahko našo odločitev potrdimo ali pa prekličemo 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 91: Sporčilno okno *AboutBox*.

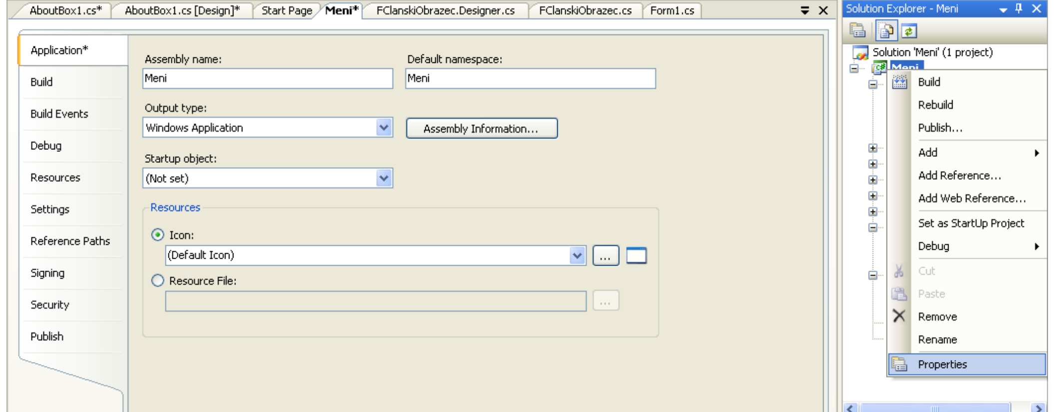
Lastnosti gradnikov *Label* in vsebino gradnika *TextBox* ne spreminjajmo, ker se bo njihova vsebina zapisala sama, oziroma jo bomo določili na drugem mestu, to je pri lastnostih projekta. Poglejmo kodo obrazca *AboutBox* (pogled *View Code*).

```
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 92: Koda obrazca *AboutBox*.

Koda se je zgenerirala avtomatično: sklicuje se kar na nekaj lastnosti, 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 93: 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.

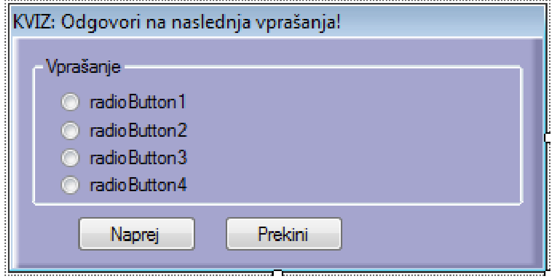
V ta namen bomo izdelali dva obrazca: glavni obrazec bo vseboval gradnik *MenuStrip* z dvema možnostima: *Začetek reševanja* (začetek reševanja kviza) in *Rezultati* (ogled dosedanjih rezultatov reševanja). V meniju naj bo še tekstovno polje, v katerega mora uporabnik pred začetkom reševanja vnesti svoje 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 94: 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 95: 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# na spošno objektov nikoli ne moremo uničiti sami. Za to opravilo ne obstaja nikakršna univerzalna 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 beri=File.OpenText(imeDatoteke))
{
    string vrstica, vsebina = "";
    //beremo dokler ne zmanjka vrstic
    while ((vrstica = beri.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 beri = File.OpenText(imeDatoteke)));
    try
    {
        string vrstica, vsebina = "";
        while ((vrstica = beri.ReadLine()) != null)
            vsebina += vrstica;
    }
    finally  //brezpogojni varovalni blok
    {
        beri.Close();
    }
}
catch
```

```
{
    MessageBox.Show("Napaka!");
}
```

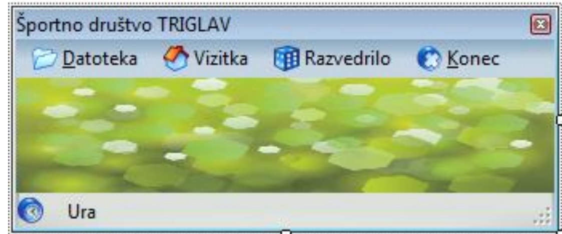
Tak način je sicer legalen in v zgornjem primeru tudi ustrezen. Problem pa nastane, kadar je potrebno odstranjevanje (sproščanje) več kot enega vira. V takem primeru bi bilo potrebno gnezdenje varovalnih blokov. Poleg tega je v zgornjem primeru referenca na objekt *branje* ostala tudi potem, ko se varovalni 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 96: 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:

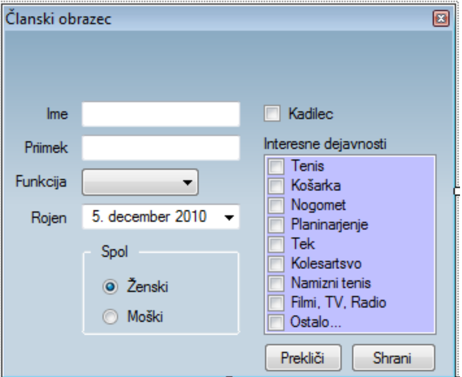
```
class Clan
{
    //javna polja razreda Clan
    public string ime, priimek;
    public string funkcija;
    public DateTime rojstvo;
    public char spol;
    public bool kadilec;
    public ArrayList dejavnosti; //netipizirana zbirka za zapis dejavnosti
    //objektna metoda, ki vrne niz z vsemi 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*,

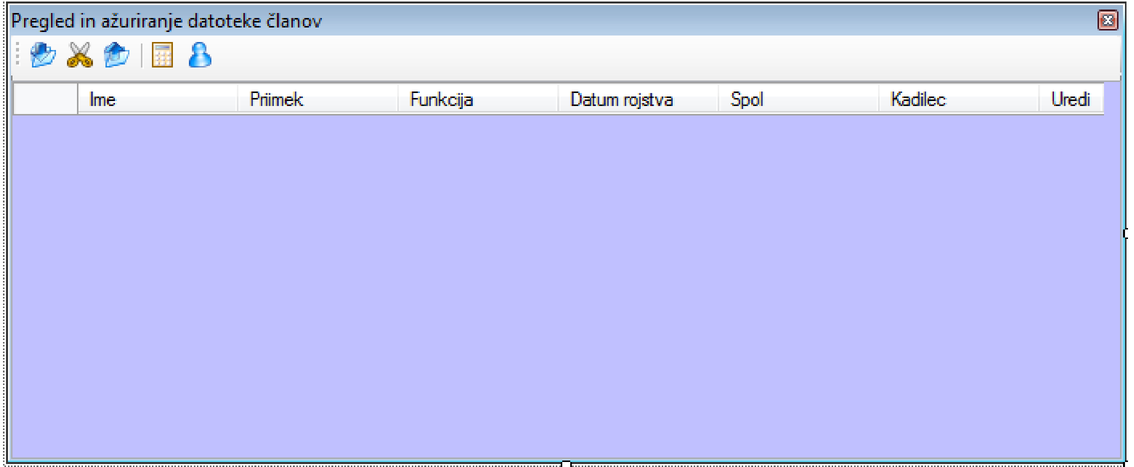
CheckedListBox in dva gumba. V gradnik *CheckBox* s pomočjo lastnosti *Items* dodajmo nekaj najpopularnejših športov.



Slika 97: 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 98: 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:

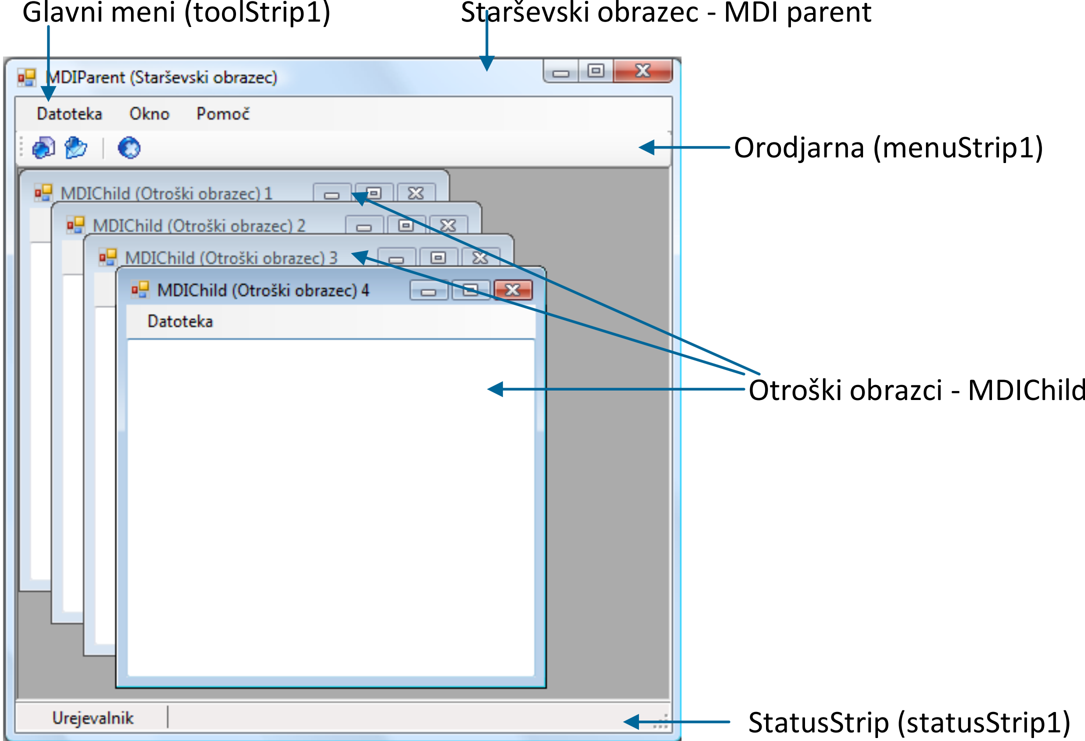
- Povezovanje obrazcev pri vmesniku *SDI* (*Single Document Interface* – vmesnik z enim samim dokumentom). Na tak način smo obrazce povezovali v vseh dosedanjih projektih. Izraz *SDI* ni najbolj posrečen, saj smo tudi pri dosedanjih obrazcih lahko imeli znotraj enega okna odprtih več dokumentov. *SDI*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

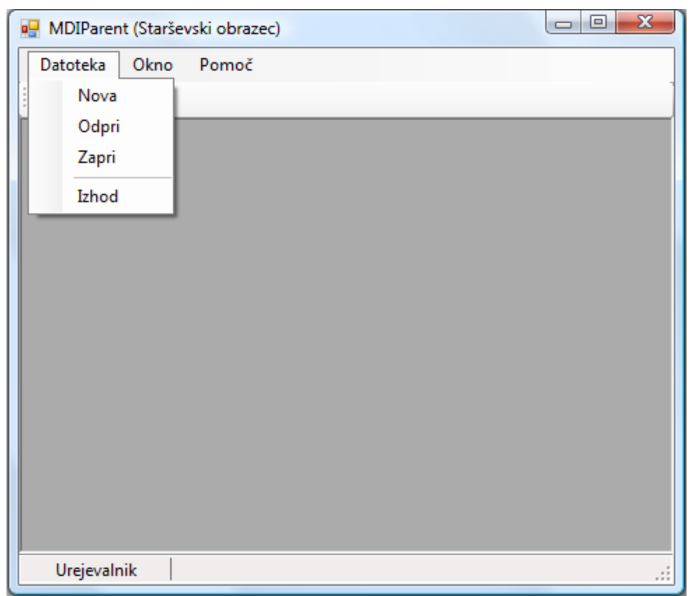
programa vsebovanih oken ne moremo postaviti izven glavnega okna. Pri takem poskusu dobi glavno okno na robovih drsnike.



Slika 99: *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 100: Starševski obrazec - *MDI Parent*.

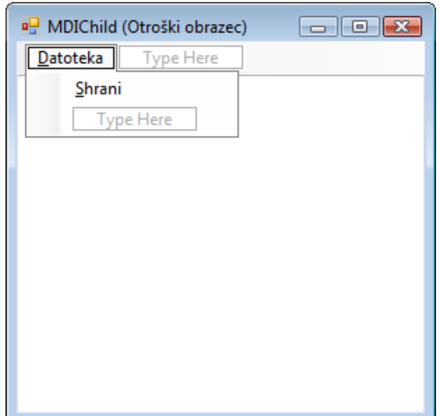
Obrazcu naj bo ime *fMDIParent*, lastnost *IsMdiContainer* postavimo na *true*. Na ta način smo obrazec označili kot starševski obrazec, ki bo vseboval otroške obrazce. V glavnem meniju naredimo tri možnosti (*Datoteka*, *Okno* in *Pomoč*), postavka *Datoteka* ima tri možnosti (*Nova*, *Zapri* in *Izhod*), 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 101: 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 102: 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();
}
```

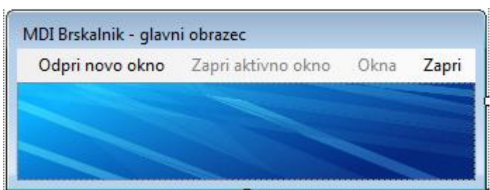
Brskalnik

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*.



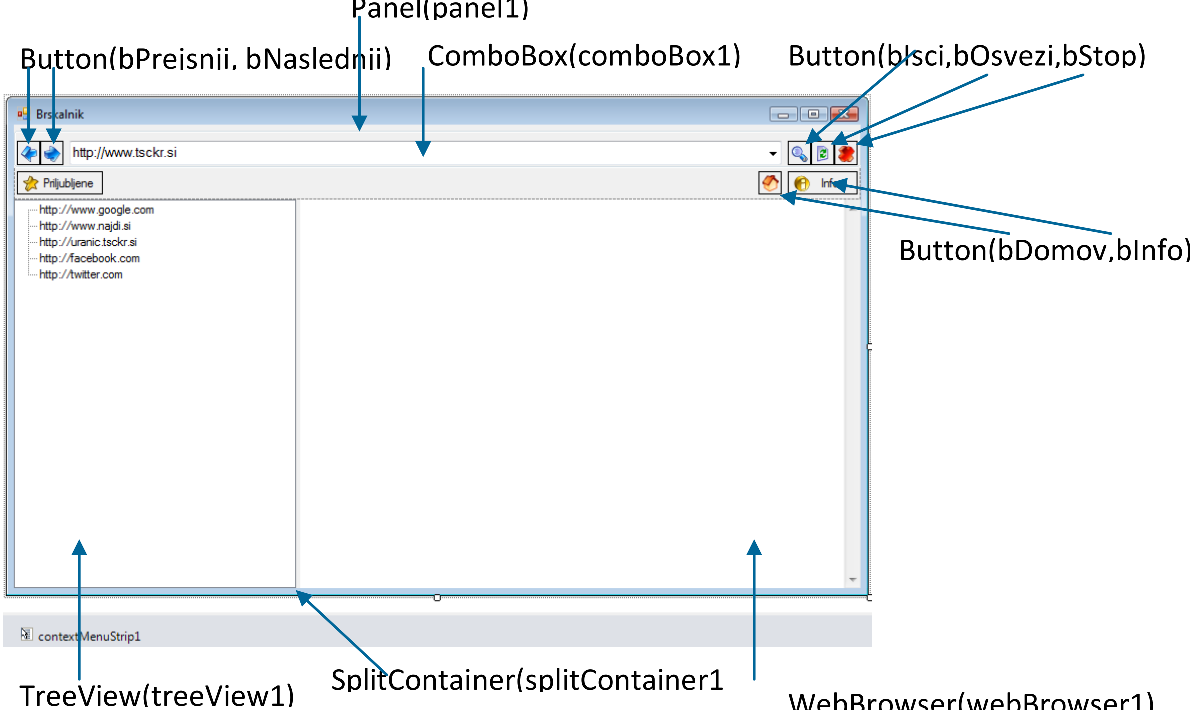
Slika 103: Glavni obrazec *MDI* aplikacije.

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 104: Lebdeči meni na otroškem obrazcu!

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



Slika 105: 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

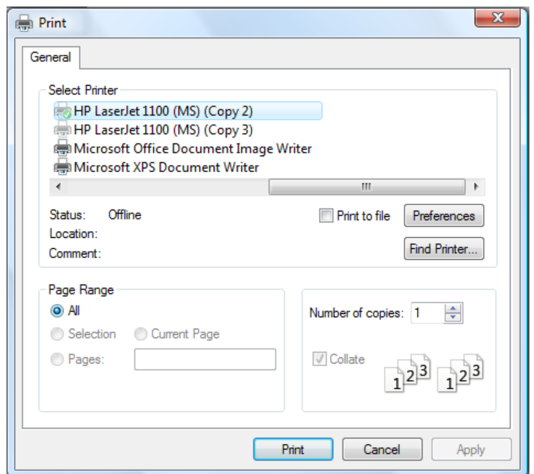
Gradnike, ki so v *Visual C#* namenjeni tiskalniškim poslom, najdemo v oknu *Toolbox* v skupini *Printing*. Gradniki so večinoma nevizuelni. Ko jih postavimo na obrazec, se prikažejo v polju pod obrazcem. Tiskamo lahko besedila in slike, večino nastavitev za tiskanje pa moramo narediti programsko.

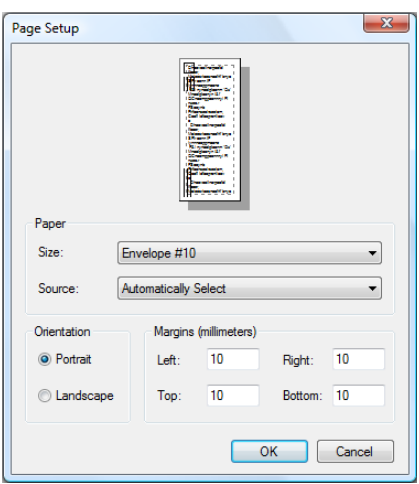
Za izbiro in nastavitev tiskalnika, ter za nastavitev strani izpisa sta na voljo dialoga

- *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 106: 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
```

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 24: 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.

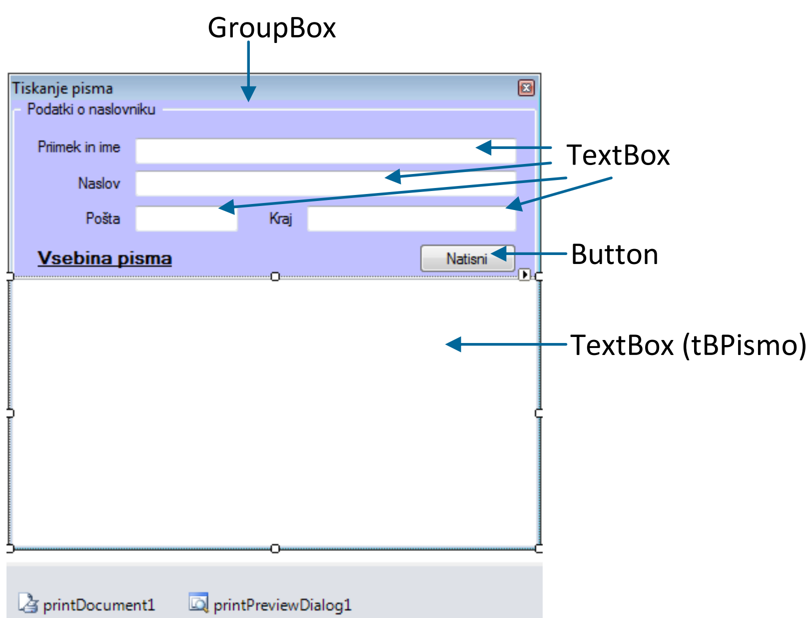
Tiskanje enostranskega besedila

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 107: 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 dogodka *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* .

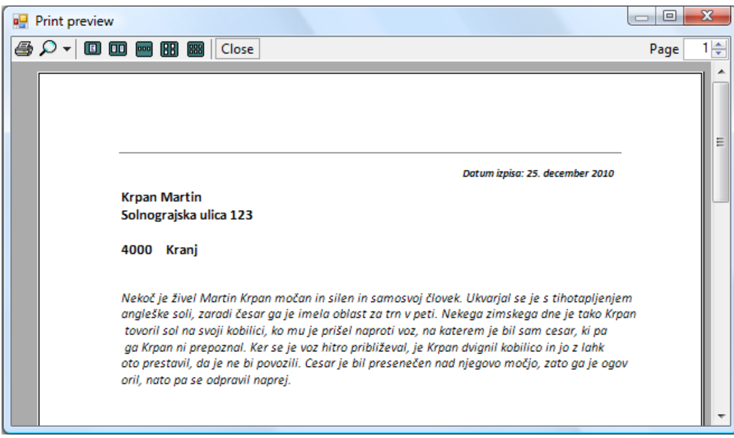
```
/*dogodek PrintPage gradnika printDocument1 – v njem zapišemo kaj in
  kako bomo tiskali*/
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    /*e.MarginBounds predstavlja pravokotno področje namenjeno tiskanju
    znotraj tiskalnikove strani*/
    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.

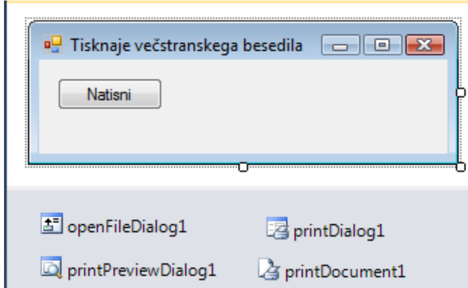


Slika 108: Predogled tiskanja.

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 109: 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.

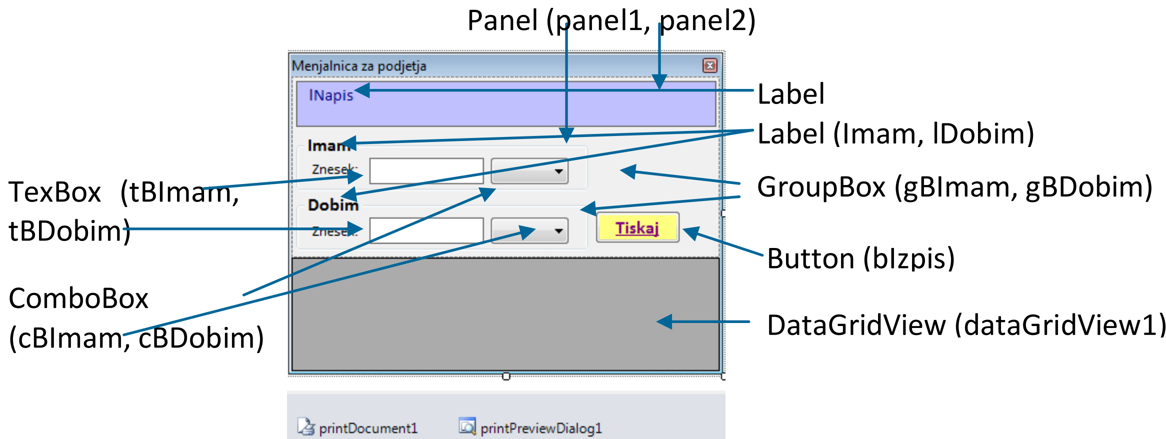
```
//Tiskanje slike: predogled
Font printFont; //pred tiskanjem bomo vsakič določili pisavo
string strPrint;//niz, v katerega bomo pisali besedilo za tiskanje
private void tiskanjeSlikePredogledToolStripMenuItem_Click(object sender,
EventArgs e)
{
    //napis na oknu
    openFileDialog1.Title= "Okno 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 110: 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.

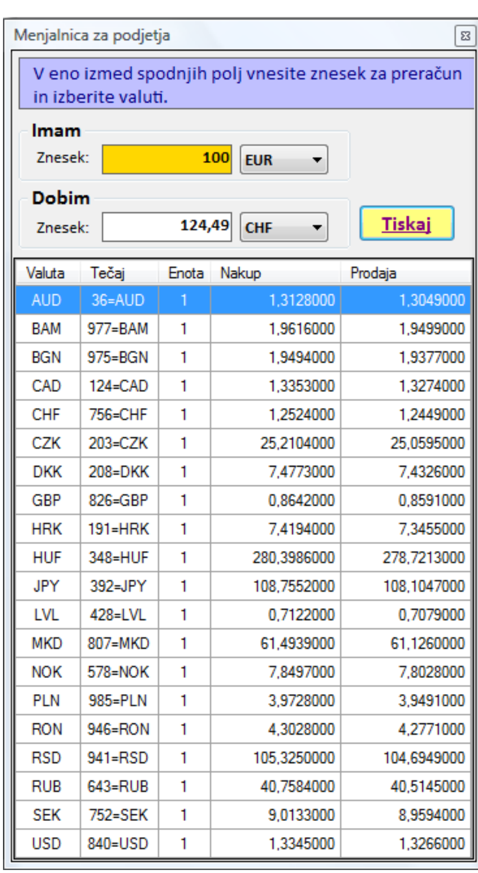
Za opravljeno menjavo želimo pripraviti tudi izpis na tiskalniku. Pred tiskanjem najprej preverimo, če bila menjava že izvedena. Izpis bi si radi ogledali najprej v predogledu.

```
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 111: Projekt *Menjalnica* v uporabi.

Kranj 31. december 2010

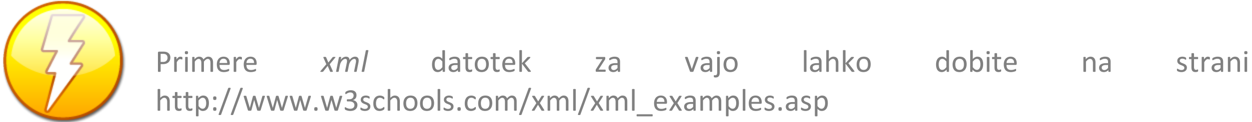
Menjalnica EURO

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 112: Projekt *Menjalnica*: primer izpisa na tiskalniku.



Primere *xml* datotek za vajo lahko dobite na strani http://www.w3schools.com/xml/xml_examples.asp

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.

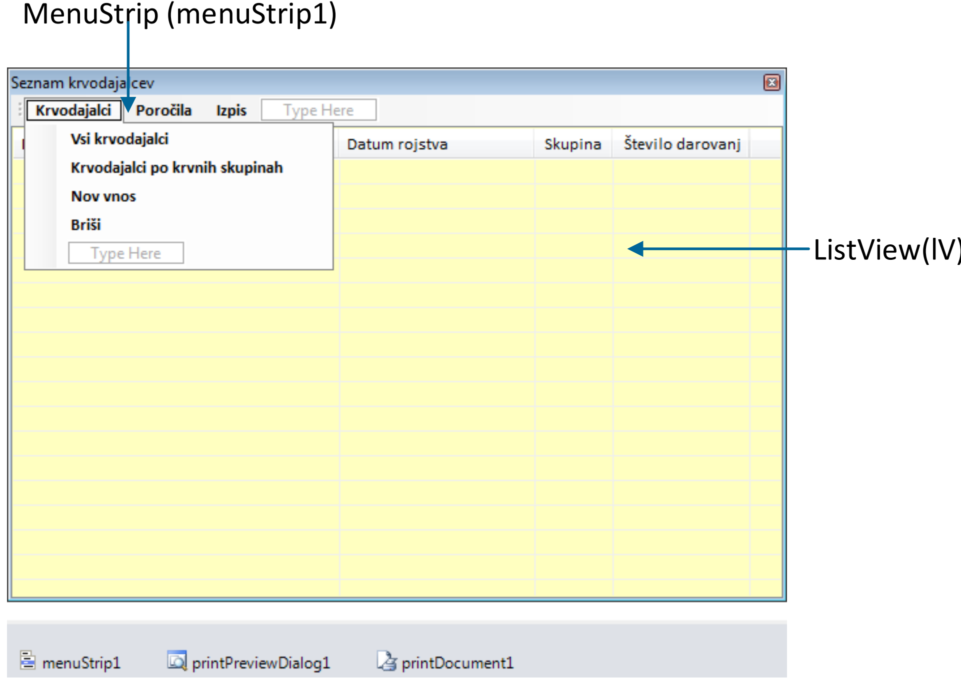
V projekt najprej vključimo nov razred. V *Solution Explorer*-ju z desnim klikom na projektno datoteko izberemo možnost *Add*, nato *New* Item in Class.*cs*. Ime datoteke spremenimo v *Razredi.cs* in vnos potrdimo s klikom na gumb *Add*. V datoteko zapišemo osnovni razred *Oseba* in iz njega izpeljimo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

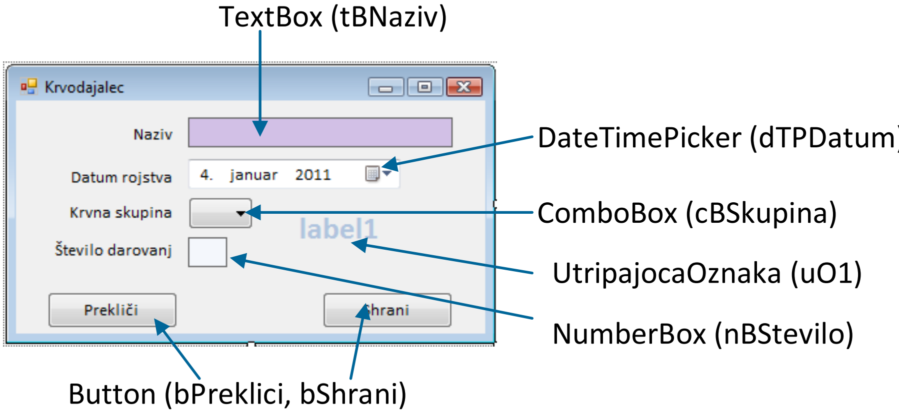
Glavni obrazec projekta poimenujmo *FSeznam*. V glavni meni na obrazcu zapišimo tri postavke (*Krvodajalci*, *Poročila* in *Izpis*). Prvi dve možnosti nato še razvejimo: postavka *Krvodajalci* naj ima štiri dodatne možnosti (glej sliko), postavka *Poročila* pa dve (*Število krvodajalcev po krvnih skupinah*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



Slika 113: 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 114: 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 *lV*. 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.u01.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);
```

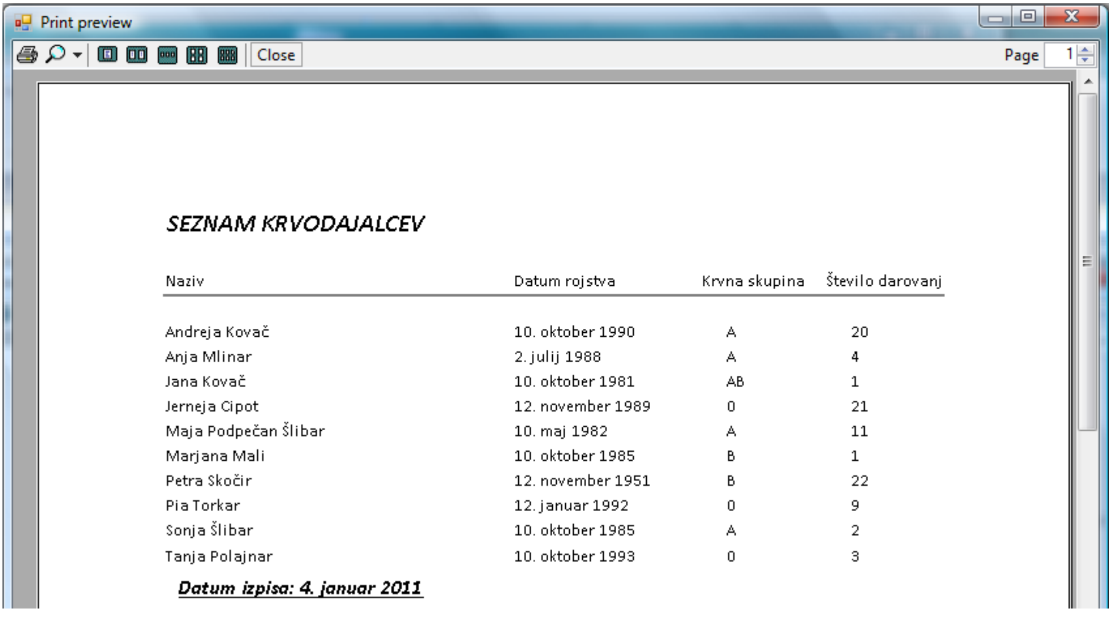
```
    int vrstica = 0; //maksimalno 45 postavk na stran
    //še izpis podatkov posameznega krvodajalca
    while (vrstica < 45 && zaporedna < lV.Items.Count - 1)
    {
        y = y + 20;
        e.Graphics.DrawString(lV.Items[zaporedna].Text,printFont,
Brushes.Black, x, y, new StringFormat());

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.



SEZNAM KRVODAJALCEV

Naziv	Datum rojstva	Krvna skupina	Število darovanj
Andreja Kovač	10. oktober 1990	A	20
Anja Mlinar	2. julij 1988	A	4
Jana Kovač	10. oktober 1981	AB	1
Jerneja Cipot	12. november 1989	0	21
Maja Podpečan Šlibar	10. maj 1982	A	11
Marjana Mali	10. oktober 1985	B	1
Petra Skočir	12. november 1951	B	22
Pia Torkar	12. januar 1992	0	9
Sonja Šlibar	10. oktober 1985	A	2
Tanja Polajnar	10. oktober 1993	0	3

Datum izpisa: 4. januar 2011

Slika 115: Primer izpisa seznama krvodajalcev.

DRŽAVE

Potrebujemo bazo podatkov z imenom *DrzaveSQL*, v njej pa dve tabeli: *Kontinenti* in *Drzave*. Pripravili bomo aplikacijo, v kateri bomo obe tabeli tudi poljubno ažurirali in dodajali nove postavke, ustvariti pa želimo tudi nekaj osnovnih poročil. Naučili se bomo izdelati SQL bazo podatkov kar znotraj razvojnega okolja, spoznali gradnike za delo z bazami podatkov, ter razrede in metode za prikaz podatkovnih tabel na obrazcih. Pojasnili bomo tudi pojem transakcije.

Podatkovna skladišča (baze) in upravljanje s podatki

V naslednjem poglavju bo razložena in prikazana manipulacija s podatkovnimi bazami. Naučili se bomo izdelati podatkovno bazo kar znotraj okolja *Visual C# Express Edition*, kasneje pa bomo uporabljali že izdelane primere podatkovnih baz, ki že vsebujejo podatkovne tabele s testnimi podatki. Za izdelavo podatkovnih baz obstajajo seveda tudi druga, bolj specializirana orodja, npr. *Microsoftov SQL Server Management Studio Express*. Za razumevanje poglavja je potrebno vsaj osnovno poznavanje relacijskih podatkovnih baz.

S prihodom *.NET* orodij, se je *Microsoft* odločil tudi za nadgradnjo svojega modela dostopa do podatkovnih baz (ActiveX Data Objects – ADO) in tako je nastal *ADO.NET*. *ADO.NET* je v bistvu množica razredov, oz. ogrodje, ki ga lahko uporabimo za interakcijo z bazami podatkov in *XML* datotekami. S pomočjo mehanizmov *ADO.NET*, je lahko povezava s podatkovno bazo uporabljena v različnih aplikacijah, pri čemer obstaja možnost začasne prekinitve povezave in ponovne vzpostavitve kar znotraj aplikacije. Tak način dela mnogokrat predstavlja občuten prihranek časa.

Izdelava baze podatkov

Razvojno *Visual C# Express Edition* omogoča enostavno izdelavo podatkovne baze in ustreznih tabel, ključev in vsega ostalega znotraj baze. Seveda nam morajo biti prej znani ključni pojmi glede baz podatkov: kaj pravzaprav baza je, čemu služi, kaj so to podatkovne tabele znotraj baze, kaj je to polje, tipi polj, pojem zapisa, ključa, …

Pri izdelavi baze kar znotraj razvojnega okolja *Visual C# Express Edition*, imamo na izbiro dve vrsti baze podatkov:

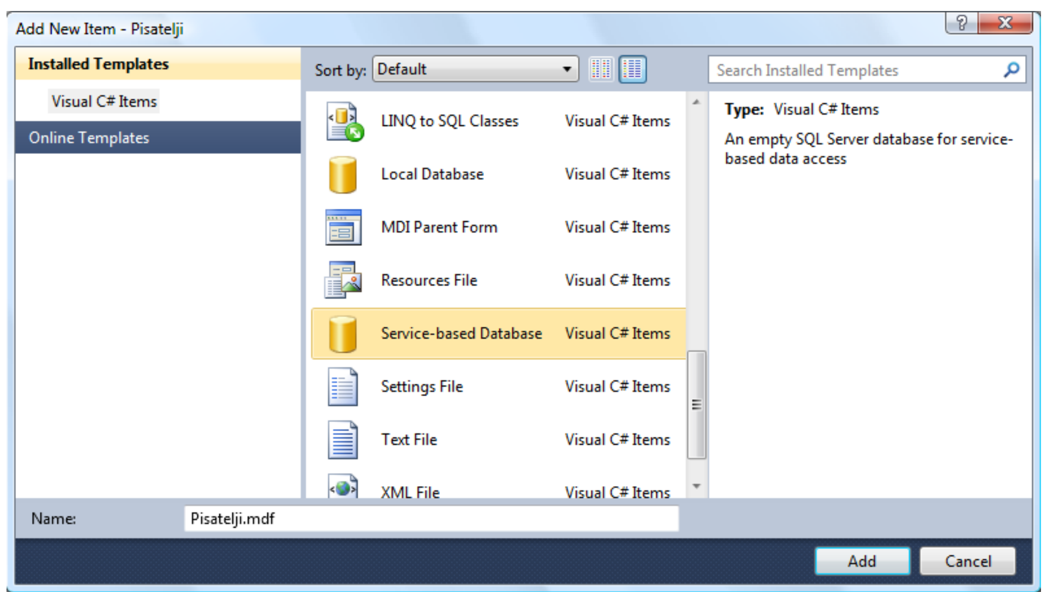
- *Local Database* in
- *Service Based Database*.

Local Database je t.i. *compact edition* baza, ki temelji na datotekah. Za neposreden dostop do podatkov v taki bazi potrebujemo le ustrezen gonilnik (*Driver*). Taka baza ne podpira t.i. *stored* procedur, ki predstavljajo zelo močan mehanizem, ki skrbi za varnost podatkov. Datoteka, v kateri je baza shranjena, ima končnico *SDF* (*SQL Server Compact Edition format*). Za dostop do take baze na lokalnem računalniku ni potrebna instalacija lokalnega strežnika.

Service Based Database pa je baza podatkov, ki je dostopna le s pomočjo *SQL* strežnika. Datoteka, v kateri je baza shranjena, ima končnico *MDF*, ki predstavlja *SQL Server format*. Za priklop na *SQL Server* bazo podatkov je na lokalnem računalniku potreben zagon *SQL Server* servisa, saj le preko tega servisa lahko dostopamo do podatkovnih tabel v bazi.

Pri izdelavi aplikacij, ki delajo s podatkovnimi skladišči, se bomo morali s pomočjo *Database Explorerja* priključiti na ustrezno bazo. V primeru, da so naše nastavitve v *SQL Server Configuration Managerju* napačne, bomo pri poskusu uspešnosti povezave na to bazo dobili obvestilo *Failed to generate a user instance of SQL Server due to failure in starting the process for the user instance*. Rešitev je naslednja: odprite *SQL Server Configuration Manager*, nato dvokliknite na *SQLServer (SQLEXPRESS)* in v spustnem seznamu *Buil-in-account* izberite *Local System*. Pobrišite (ali pa bolje le preimenujte) mapo C:\Users\[*user*]\AppData\Local\Microsoft\Microsoft SQL Server Data**SQLEXPRESS**

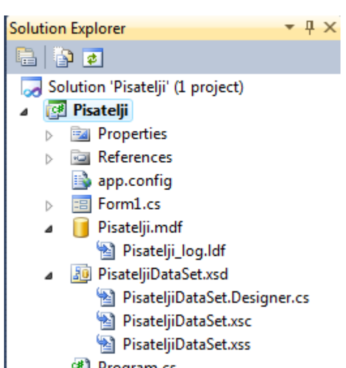
Za začetno vajo izdelajmo nov projekt in ga poimenujmo *Pisatelji*. Novo bazo, ki jo bomo v tem projektu uporabili, ustvarimo tako, da v meniju *Project izberemo Add New Item…* V oknu izberimo *Service-based Database*, bazo poimenujmo *Pisatelji.mdf* in kliknimo *Add*.



Slika 116: Izdelava nove baze podatkov.

V oknu *Solution Explorer* se pojavi nova postavka *Pisatelji.mdf*, na ekranu pa se čez nekaj časa pojavi okno *Data Source Configuration Wizard*. Razvojno okolje nam ponudi dva podatkovna

modela (<u>*POZOR*</u>: izbira dveh modelov je možna šele pri verziji 2010, pri prejšnjih verzijah pa je model *Dataset* privzet!). Izberimo *Dataset* in kliknimo gumb *Next*. Ker želimo ustvariti novo bazo, se čez nekaj časa v oknu pojavi obvestilo, da ustvarjamo novo bazo, oz. da baza, ki jo izdelujemo še ne vsebuje objektov. Razvojno okolje bo zato za nas ustvarilo objekt tipa *DataSet*, to je objekt, ki predstavlja nekakšno kopijo naše baze v pomnilniku. V tem objektu bodo začasno shranjeni vsi podatki iz tabel, ki pripadajo bazi podatkov. Ime objekta (v našem primeru *PisateljiDataSet*) se pojavi v spodnjem levem delu okna, s klikom na gumb *Finish* pa ime potrdimo. Ustvarili smo prazen *DataSet*, ki se pojavi tudi v oknu *Solution Explorer*.

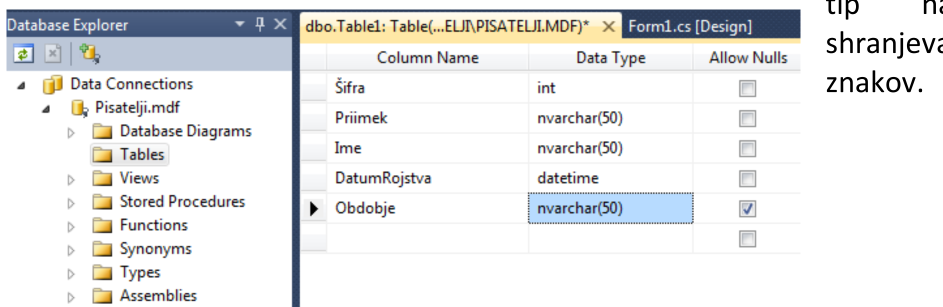


Slika 117: Okno *Solution Explorer* in v njem objekt tipa *DataSet* z imenom *PisateljiDataSet*

V naslednjem koraku bomo v bazi *Pisatelji* ustvarili še tabelo z imenom *Pisatelji*. Dvokliknimo na datoteko *Pisatelji.mdf* v *Solution Explorerju* in (običajno na levi strani) se nam odpre novo okno *Database Explorer*. Namenjeno je ustvarjanju novih tabel in ažuriranju že obstoječih.

Slika 118: Okno *DataBase Explorer*.

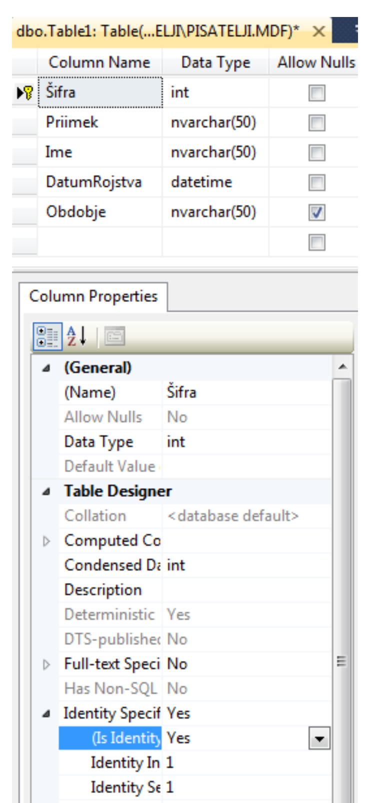
V oknu *Database Explorer* izberimo vrstico *Tables*, kliknimo desni miškin gumb in nato *Add New Table*. Odpre se urejevalnik polj za novo tabelo. Pisatelja bomo opisali s polji, ki so prikazni na naslednji sliki. Dodajmo še, da je razlika med poljema tipa *nvarchar* in *varchar* v tem, da je prvi tip namenjen tudi za shranjevanje neangleških znakov.



Slika 119: Ustvarjanje polj v tabeli *Pisatelji*.

Vsem poljem (razen polju *Obdobje*) smo odstranili kljukico v stolpcu *Allow Nulls*. S tem smo prepovedali prazne vnose posameznih polj v tabelo. Z miško se postavimo še v polje *Šifra* in kliknimo desni gumb. V oknu, ki se prikaže, izberimo *Set Primary Key*. Nastavili smo t.i. primarni ključ, kar v našem primeru pomeni, da se vrednost polja *Šifra* v tabeli ne more nikoli ponoviti. Polju *Šifra* določimo še lastnost *Auto Increment*. Za polja, ki imajo to lastnost je značilno, da bo *SQL Server* njihovo vrednost določal avtomatično. Postopek je naslednji: izberimo polje *Šifra*, nato pa v oknu *Column Properties*, ki se nahaja pod oknom za urejanje polj, nastavimo lastnost

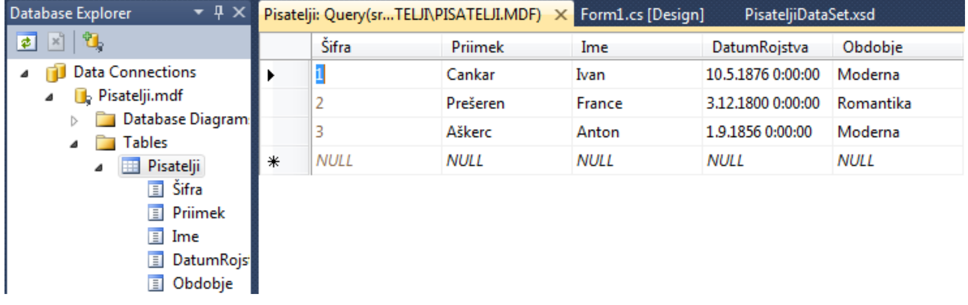
Identity Specification. Podlastnost *Is Identity* nastavimo na *Yes*, podlastnosti *Identity Increment* (vrednost, za katero se bo polje avtomatično povečevalo) in *Identity Seed* (izhodiščna vrednost) pa nastavimo po želji. Privzeti vrednosti sta enaki 1. V našem primeru smo pustili vrednosti privzeti, kar pomeni, da bo šifra prvega pisatelja v tabeli enaka 1, vsem ostalim pa se bo vrednost avtomatično povečevala za 1.



Slika 120: Okno *Column Properties*.

Končno se ponovno postavimo z miško na zavihek nad urejevalnikom polj. Kliknimo desni miškin gumb, izberimo opcijo *SaveTable1* in za ime tabele vnesemo *Pisatelji*. Ustvarjanje prve tabele znotraj naše baze podatkov *Pisatelji* je tako končano.

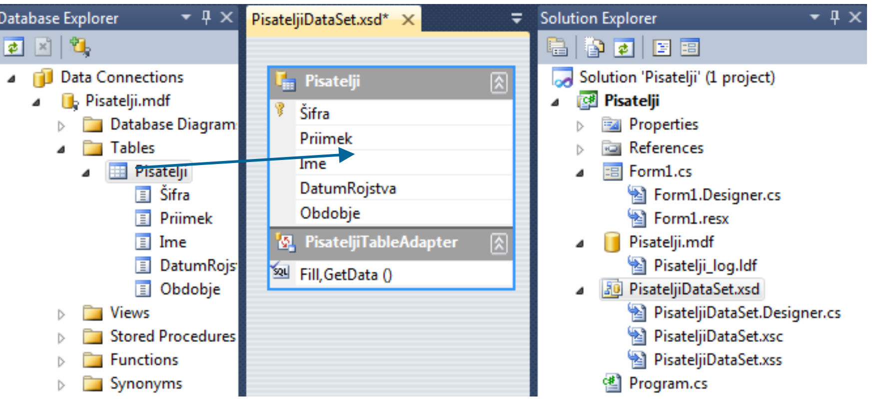
Sedaj lahko pričnemo z vpisovanjem testnih podatkov v to tabelo. V *DataBase Explorerju* razširimo vrstico *Tables* pod bazo *Pisatelji*.mdf, napravimo desni klik na tabelo *Pisatelji* in izberimo opcijo *Show Table Data*. V tabelo, ki se prikaže, lahko že kar na tem mestu vnesemo nekaj testnih podatkov. Šifer seveda ne vnašamo, ker smo polju nastavili lastnost *Autoincrement*, se vrednosti temu polju dodeljujejo avtomatično, od 1 naprej.



Slika 121: Vnos podatkov v tabelo *Pisatelji*.

Kasnejše ažuriranje polja neke tabele znotraj razvojnega okolja *Express Edition* NI možno. Če torej hočemo v shranjeni tabeli kakorkoli spremeniti poljubno obstoječe polje (npr povečati število znakov), bomo pri ponovnem shranjevanju dobili obvestilo, da to ni možno. Seveda pa lahko poljubno polje v tabelo kadarkoli dodamo ali pa ga pobrišemo in nato tabelo ponovno shranimo.

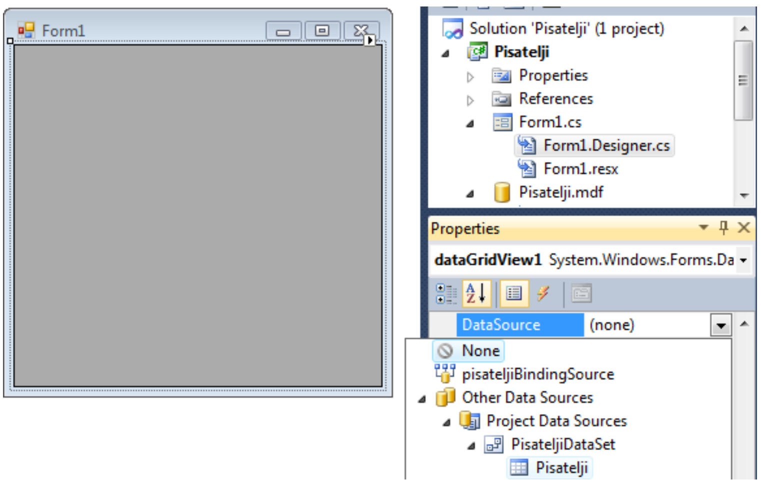
Pri ustvarjanju zgornje baze *Pisatelji* in v njej tabele *Pisatelji*, nam je Visual C# ustvaril tudi prazen *DataSet* in ga poimenoval *PisateljiDataSet*. Ime smo pustili kar privzeto. V ta *DataSet* bomo sedaj dodali našo tabelo *Pisatelji* in jo tako naredili dostopno gradnikom na obrazcu. V oknu *Solution Explorer* dvokliknimo na *PisateljiDataSet*, nato pa v okno, ki se odpre, povlecimo našo tabelo *Pisatelji*.



Slika 122: Tabela *Pisatelji* v gradniku *DataSet*.

Tabela je sedaj pripravljena in do nje lahko dostopamo tudi preko lastnosti gradnikov. Ogledamo si lahko tudi predogled vsebine podatkovne tabele *Pisatelji*. V oknu *PisateljiDataSet* desno kliknemo na tabelo *Pisatelji*. Odpre se meni, v katerem izberemo možnost *Preview Data*. Ko se okno *Preview Data* odpre, v njem le še kliknemo gumb *Preview*.

Vsebino tabele *Pisatelji* bomo sedaj prikazali npr. v gradniku *DataGridView*. Na zaenkrat še prazen obrazec našega projekta, ki smo ga na začetku poimenovali *Pisatelji*, dodajmo gradnik *DataGridView*. Ime objekta naj bo kar privzeto, to je *dataGridView1*. Lastnost *Dock* mu natavimo na *Fill*, da se razširi čez celoten obrazec. V tem objektu bi sedaj radi prikazali vsebino tabele *Pisatelji* naše baze *Pisatelji*. Postopek je sledeč: izberimo objekt *dataGridView1* in v oknu *Properties* lastnost *DataSource*. V spustnem seznamu izberemo *Other Data Sources*. Kliknemo na znak +, da se seznam razširi, nato razširimo še *ProjectDataSources*, *PisateljiDataSet* in končno izberemo *Pisatelji*.



Slika 123: Povezava gradnika *DataGridView* s podatkovnim izvorom – tabelo v bazi podatkov.

V objektu *dataGridView1* so nastali stolpci, ki so dobili enaka imena kot so polja v tabeli *Pisatelji*. Napise na vrhu stolpcev lahko kadarkoli poljubno preimenujemo. To storimo na enak način, kot smo to spoznali v poglavju *Gradnik DataGridView*.

Na obrazcu je vsebina tabele *Pisatelji* zaenkrat še nevidna, a če projekt zaženemo, je *dataGridView1* napolnjen z ustreznimi podatki tabele *Pisatelji*. Razvojno okolje je ob tem ustvarilo odzivno metodo *Form1_Load* dogodka *Load* našega obrazca in vanj zapisalo stavek.

```
this.pisateljiTableAdapter.Fill(this.pisateljiDataSet.Pisatelji);
```

S pomočjo tega stavka podatke iz tabele *Pisatelji* prenesemo v ustrezen objekt v pomnilniku. Ta objekt smo nato povezali z objektom gradnika *DataGridView*, ki podatke prikaže na obrazcu.

Ko smo gradili projekt je Visual C# dodal nekaj nevizuelnih gradnikov. Ti so vidni v polju pod obrazcem, njihov pomen in uporabo bomo spoznali v nadaljevanju.

Seveda nas zanima še to, v kateri mapi je razvojno okolje ustvarilo novo bazo, pa seveda tudi to, kako pravkar izdelano bazo prestaviti v kako drugo mapo. Lokacija baze je v fazi načrtovanja projekta zapisana v datoteki *App.config*, ki se nahaja v mapi z našim projektom. Običajno jo najdemo tudi v oknu *Solution Explorer*. V tej datoteki je tudi stavek *connectionString* z vsemi potrebnimi podatki o lokaciji in dostopu do baze podatkov.

```
connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\Pisatelji.mdf;Integrated
Security=True;User Instance=True"
```

Baza se torej na začetku, potem, ko smo jo ustvarili, nahaja v ***|DataDirectory|\Pisatelji.mdf***. V V tej mapi je tudi datotka *app.config* in ostalime glavne datoteke našega projekta. Ko pa projekt z vključeno bazo prvič prevedemo, razvojno okolje celotno bazo prekopira v mapo z izvršilno datoteko našega projekta, to je v mapo ..\bin\Debug. Razvojno okolje v isti mapi ustvari tudi konfiguracijsko datoteko, ki ima enako ime kot izvršilna datoteka in še dodatno končnico *.config*. V zgornjem primeru je ime datoteke *Pisatelji.exe.config*. Bazo sestavljata dve datoteki, *Baza.mdf* in *Baza_log.ldf*, v zgornjem primeru torej *Pisatelji.mdf* in *Pisatelji_log.ldf.* Če želimo bazo prestaviti v drugo mapo, moramo ti dve datoteki najprej premakniti v želeno mapo, nato pa spremeniti *connectinString* v konfiguracijski datoteki, npr. takole:

```
connectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=c:\BAZE\Pisatelji.mdf;Integrated
Security=True;User Instance=True"
```

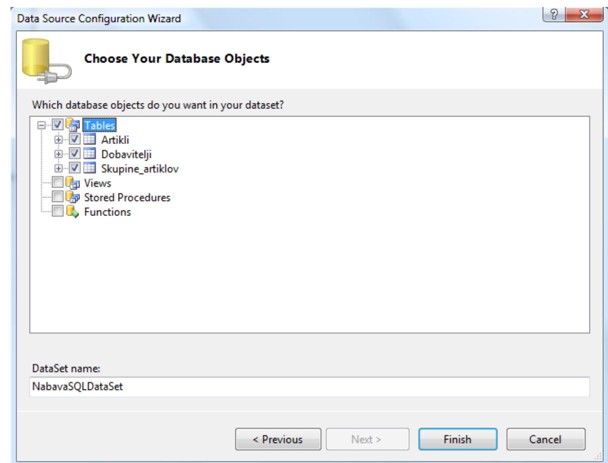
V fazi načrtovanja našega projekta bo ta še vedno uporabljal lokalno bazo v mapi z glavnimi datotekami našega projekta. Pri prevajanju se bo baza še vedno prekopirala v mapo *bin\Debug*, aplikacija pa bo ob zagonu dejansko uporabljala bazo, katere pot je napisana v konfiguracijski datoteki. To seveda pomeni, da moramo pri nameščanju aplikacije, ki dela z bazo podatkov, na ciljni računalnik, namestiti tako izvršilno datoteko, kot tudi konfiguracijsko datoteko, pa seveda tudi ustrezno bazo podatkov. Dobra stran tega dejstva pa je, da je lokacija baze neodvisna od izvršilnega programa, ki ga zaradi tega ni potrebno spreminjati ne glede na to v kateri mapi ali na katerem strežniku je baza.

Izdelava novega projekta, ki dela nad obstoječo bazo podatkov – uporaba čarovnika

Pri ustvarjanju novega projekta navadno uporabljamo že obstoječo bazo podatkov, bodisi smo jo naredili že kdaj prej, ali pa jo je za nas naredil že kdo drug. V naslednjem primeru bomo uporabili že obstoječo bazo *NabavaSQL* s tremi tabelami, ki je dostopna na strežniku *http://uranic.tsckr.si/Lokalne%20baze/*. Bazo si najprej prekopirajmo v mapo na svojem računalniku, nato pa ustvarimo nov projekt z imenom *Nabava*. Bazo *NabavaSQL* lahko v naš projekt s pomočjo čarovnika vključimo na dva načina:

- V *Solution Explorerju* označimo naš projekt, kliknemo desni gumb miške, izberemo opcijo *Add* in nato *Add Existing Item*. Odpre se okno za dodajanje obstoječe datoteke v projekt. V spustnem seznamu na dnu tega okna določimo vrsto datoteke, ki jo bomo dodali v projekt. Izberemo *Data Files* - datoteke tipa *.mdf* in izberemo ustrezno datoteko. Če smo le-to prej skopirali v mapo z našim projektom jo le izberemo, sicer pa se premaknemo v ustrezno mapo na disku, ki vsebuje to bazo podatkov in jo izberemo. S klikom na gumb *Add* jo vključimo v naš projekt. V našem primeru v projekt vključimo bazo *NabavaSQL*. Bazo smo na ta način prekopirali v mapo našega projekta, tako da v fazi načrtovanja oz. izdelave projekta delamo s kopijo prave baze!

  Čez nekaj časa se prikaže okno *Data Source Configuration Wizzard*, kjer izberemo *Dataset* in kliknemo gumb *Next*. Prikaže se seznam tabel, ki pripadajo izbrani bazi podatkov. Odkljukajmo tiste, ali pa vse, s katerimi bomo v projektu delali in kliknimo gumb *Finish*.

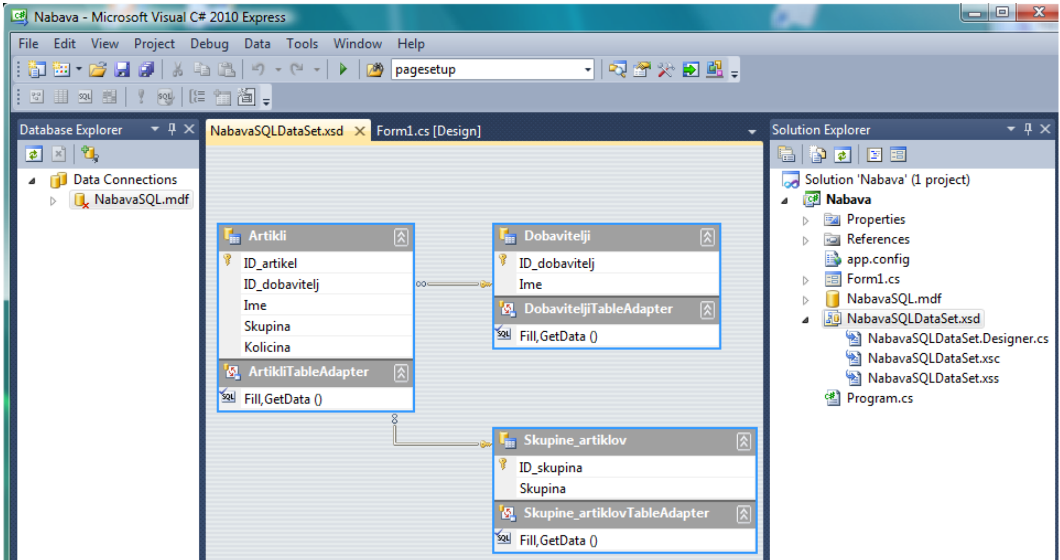


Slika 124: V oknu *Data Source Configuration Wizard* izberemo tabele, ki jih bomo potrebovali.

Vsebina stavka connectionString je specifična za posamezno bazo podatkov. Na strani http://www.connectionstrings.com/ so prikazani najpomembnejši povezovalni nizi za posamezne baze.

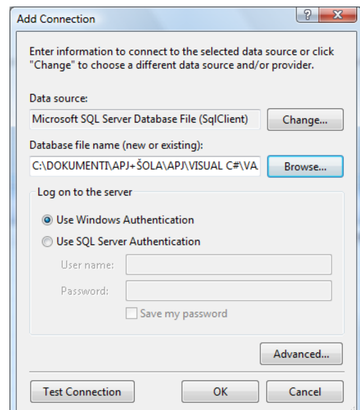
Razvojno orodje bo ob tem za nas ustvarilo gradnik *Dataset* z imenom *NabavaSQLDataSet*, ki se prikaže v oknu *Solution Explorer*. Če nanj dvokliknemo, dobimo

grafični prikaz, obenem pa še povezave med tabelami. Te so odvisne od primarnih ključev v tabelah. V *Solution Explorerju* se pojavi tudi datoteka *app.config*, ki vsebuje podatke o povezavi do baze podatkov (*connectionString*) v fazi načrtovanja projekta. Pri prvem prevajanju projekta bo v mapi *...Bin/Debug* nastala konfiguracijska tekstovna datoteka s končnico *.exe.config*. V njej bo zapisana pot, ki jo bo uporabljal izvršilni program. Pri prenosu na drug računalnik je potrebno le v tej datoteki spremeniti pot do baze na novem računalniku.



Slika 125: *Dataset* z vključenimi tabelami iz baze *NabavaSQL*.

- **Drugi način**: V meniju *Project* izberemo *ADD New Item*, zatem *Dataset*, ki ga poimenujemo npr. *NabavaDataSet* in kliknemo gumb *Add*. V urejevalniškem delu se prikaže okno *Dataset Designer*. V oknu kliknemo na označen tekst *Database Explorer*, da se na levi strani prikaže okno *Database Explorer*. Z miško se postavimo v okno *Database Explorer* in desno kliknimo. V primeru, da z bazo delamo prvič, se nam prikaže okno, v katerem izberemo opcijo *Add Connection...* Odpre se okno *Add Connection*.



Slika 126: Okno *Add Connection*.

Za podatkovni izvor (*DataSource*) izberemo *Microsoft SQL Server Database File*. S klikom na gumb *Browse* poiščemo bazo podatkov, v našem primeru *NabavaSQL.mdb*. Po želji še preverimo, če povezava z bazo deluje (klik na gumb *Test Connection*), s klikom na gumb *OK* pa okno zapremo.

V oknu *DataBase Explorer* kliknemo na *NabavaSql.mdf→Tables* in v okno *NabavaSQLDataSet* povlečemo vse tri tabele iz baze.

Če pa smo z bazo že delali (npr. v prejšnjem projektu), se prikaže okno *TableAdapter Configuration Wizard*. V njem izberemo obstoječo povezavo z bazo, ali pa s klikom na gumb *New Connection* odpremo okno *Add Connection.* Izbiro potrdimo s klikom na gumb *OK*, nato pa le še sledimo navodilom, ki nam jih ponuja čarovnik. Tudi pri tem, drugem načinu, se v oknu *Solution Explorerj* pojavi datoteka *app.config*, ki vsebuje podatke o povezavi do baze podatkov v času načrtovanja projekta (*connectionString*).

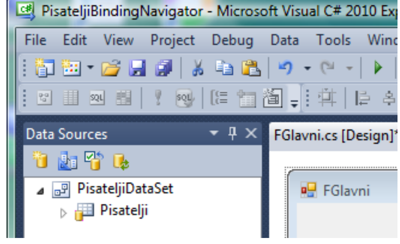
Ko smo na enega od načinov bazo vključili v projekt, lahko podatke izbrane tabele baze podatkov prikažemo npr. v gradniku *DataGridView*. Gradnik postavimo na obrazec, kliknemo na ikono v desnem zgornjem kotu tega gradnika in odpremo spustni seznam *Choose Data Source*. Do lastnosti Choose Data Source pridemo tudi tako, da izberemo gradnik *DataGridView1* in v oknu *Properties* odpremo spustni seznam *DataSource*. Razširimo *Other Data Source→Project Data Sources→NabavaSQL* in končno izberemo npr. tabelo *Artikli*! Posebnost podatkov, prikazanih v gradniku *DataGridView* je tudi v tem, da jih lahko urejamo po poljubnem stolpcu, potreben je le klik v celico na vrhu stolpca!

OPOZORILO: v fazi razvoja projekta so v gradniku *DataGridView* prikazana le imena stolpcev tabele iz baze podatkov, vsebina pa se pokaže šele ko projekt zaženemo!

Gradnik BindingNavigator in gradnik TableAdapterManager

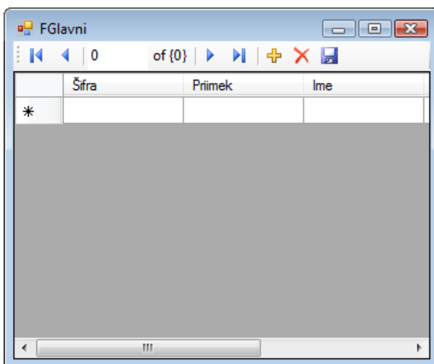
Kadar obstoječo bazo v ključimo v projekt s pomočjo čarovnika, lahko ažuriranje tabel in vstavljanje novih zapisov realiziramo s pomočjo gradnikov *BindingNavigator* in *TableAdapterManager*. Gradnika omogočata enostavno premikanje po podatkovni tabeli, in enostavno dodajanje oziroma brisanje zapisov.

Pripravimo projekt, v katerem bomo lahko v bazo *Pisatelji* dodajali nove zapise in le-te urejali s pomočjo gradnika *BindingNavigator*. Projekt poimenujmo *PisateljiBindingNavigator*. S pomočjo čarovnika v projekt najprej vključimo bazo *Pisatelji*, ustrezen objekt tipa *DataSet* pa naj ima ime *PisateljiDataSet*. Dvokliknimo na *PisateljiDataSet* v oknu *SolutionExplorer*, da se na levi strani odpre okno *DatabaseExplorer*, nato pa v meniju *Data* izberimo *DataSources*. Na levi strani se odpre dodatno okno *Data Sources*, v njem pa vidimo našo tabelo *Pisatelji*.



Slika 127: Okno *Data Sources*.

Tabelo sedaj z miško potegnimo na obrazec in razvojno okolje nam na obrazcu prikaže gradnik *BindingNavigator*, pod njim pa podatke iz tabele *Pisatelji*, ki so prikazani v gradniku *DataGridView*. Gradniku *DatagridView* nastavimo lastnost *Dock* na Fill.

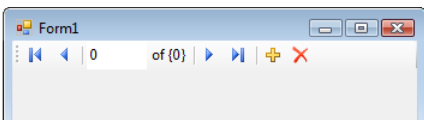


Slika 128: Gradnika *BindingNavigator* in *DataGridView*.

Navigator omogoča premikanje po tabeli, dodajanje novih zapisov na dnu tabele in shranjevanje novih zapisov v bazo podatkov. Pri vnosu novih podatkov moramo le-te vnesti pravilno, oziroma morajo biti vnosi veljavni, sicer bo pri shranjevanju prišlo do napake.

V zgornjem primeru smo si pri postavljanju gradnika *BindingNavigator* na obrazec pomagali s čarovnikom. Seveda pa lahko podoben obrazec ustvarimo tudi brez uporabe čarovnika. Odprimo projekt *Pisatelji*, ki smo ga ustvarili na začetku poglavja o bazah. Na obrazcu izberimo gradnik *dataGridView1* in ga začasno odstranimo (pobrišimo) iz obrazca. To storimo zato, ker bomo na obrazec najprej postavili gradnik *BindingNavigator* in šele nato zopet *DataGridView*, da bo le-ta zavzemal celotno površino pod gradnikom *BindingNavigator*.

Gradnik *BindingNavigator* je orodjarna z gumbi za premikanje po podatkovni tabeli in standardnimi urejevalniškimi gumbi. Nahaja se na paleti *Data*. Ko ga postavimo na obrazec, se prilepi na vrh obrazca, v njegovi orodjarni pa so le osnovni, standardni gumbi.

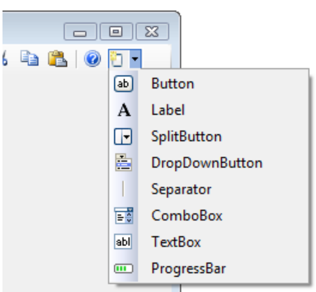


Slika 129: Gradnik *BindingNavigator*.

Dodatne gumbe vljučimo takole: izberemo gradnik (*BindingNavigator*), kliknemo na trikotnik v desnem zgornjem kotu in izberemo opcijo *Insert Standard Items*. V orodjarni se prikažejo vsi standardni urejevalniški gumbi.

Slika 130: *BindingNavigator* z vsemi standardnimi urejevalniškimi gumbi.

Poljuben gumb lahko iz orodjarne odstranimo, lahko ga izrežemo, ali pa kopiramo. Najprej ga izberemo, kliknemo desni miškin gumb in iz nato izberemo eno od možnosti *Delete*, *Cut* ali pa *Copy*. Dodajamo lahko tudi nove gumbe, labele in ostale gradnike, ki so na voljo. Potrebno je le odpreti spustni seznam *AddToolStripButton*, ki se nahaja v zgornjem desnem kotu gradnika.



Slika 131: Dodajanje novih gumbov v *BindingNavigator*.

Nadaljujmo sedaj z gradnjo našega projekta. Na obrazec postavimo nazaj gradnik *DataGridView* in mu lastnost *Dock* nastavimo na *Fill*. Povežimo ga s podatkovnim izvorom Pisatelji tako, da za *DataSource* izberemo *Pisatelji*. Urejevalniške gumbe gradnika *BindingNavigator* aktiviramo tako, da tudi tega povežemo z istim podatkovnim izvorom. Izberemo gradnik *BindingNavigator*, ki je že na obrazcu, v oknu *Properties* izberemo lastnost *BindingSource* in izberemo isti podatkovni izvor kot smo ga prej pri gradniku *DataGridView* (v našem primeru *pisateljiBindingSource*). Gumbi za premik po tabeli, dodajanje in brisanje zapisov postanejo aktivni.

Dodajanje novega zapisa v gradnik *DataGridView* lahko izvedemo na dva načina:

- Uporabimo obstoječi gumb za dodajanje, torej gumb, ki je sestavni del gradnika *BindingNavigator*. Ko projekt prevedemo, se ob kliku na ta gumb na dnu gradnika *DataGridView* že pokaže nova prazna vrstica.
- V gradniku *BindingNavigator* ustvarimo nov gumb, mu določimo ustrezno sliko, v odzivno metodo dogodka *Click* tega gumba pa zapišemo programsko kodo za dodajanje nove vrstice v gradnik *DataGridView* takole:

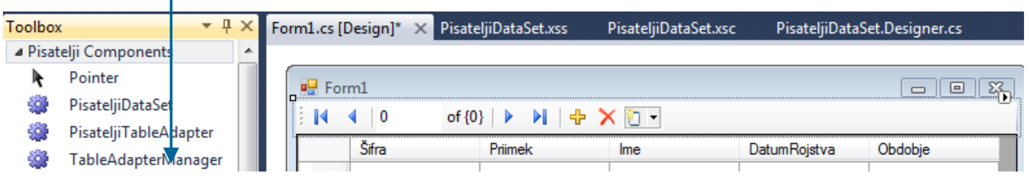
```
pisateljiBindingSource.AddNew();
```

Podatke za nov zapis sedaj vnesemo neposredno v celice gradnika *DataGridView*, ali pa za ta namen ustvarimo nov obrazec z ustreznimi gradniki (kar je bolje). Uporabnik bo vnesel podatke, ki jih bomo lahko validirali in šele nato programsko zapisali v celice gradnika *DataGridView*.

Pri spreminjanju, dodajanju ali brisanju podatkov v gradniku *DataGridView* moramo biti zelo pozorni. Če hočemo, da bodo spremenjeni podatki ažurirani tudi v bazi podatkov, moramo obvezno ažurirati še bazo. To storimo z uporabo gradnika *TableAdapterManager*. Ta se pojavi v oknu *Toolbox* šele potem, ko smo z uporabo čarovnika ustvarili povezavo z neko tabelo v bazi podatkov. Na obrazec ga postavimo tako kot ostale gradnike, le da je ta gradnik neviden in z njim le dobimo dostop do pomembnih razredov in metod za delo s podatkovnimi tabelami.

Kadar imamo na obrazcu več gradnikov, ki so povezani z nekim podatkovnim izvorom (tako kot v zgornjem primeru gradnik *BindingNavigator* in *DataGridView*), morata imeti oba gradnika isti podatkovni izvor (*BindingSoure*). Le tako bo med njima obstajala povezava.

Gradniku *TableAdpterManager*, ki smo ga postavili na obrazec, pustimo ime kar privzeto.



Slika 132: Gradnik *TableAdapterManager*.

Najpomembnejša metoda gradnika je metoda *UpdateAll*, s katero ažuriramo tabelo v bazi podatkov. V *BindingNavigator* najprej vstavimo dodatni gumb. Gumbu napišemo odzivno metodo dogodka *Click*, v njej pa s pomočjo metode *Update* podatke ažuriramo tudi v bazi podatkov. Pred tem z metodo *Validate* preverimo veljavnost zadnjega vnosa, z metodo *EndEdit* pa spremembe zapišemo v pripadajoči podatkovni izvor.

```
//odzivna metoda dogodka Click gumba Shrani v gradniku BindingNavigator
private void saveToolStripButton_Click(object sender, EventArgs e)
{
    /*Metoda Validate preveri veljavnost polja, ki je trenutno aktivno (ima
      focus) in v primeru napake izvrže izjemo!*/
    Validate();

    pisateljiBindingSource.EndEdit();/*Zaključek urejanja, zadnje spremembe
                                       se zapišejo v ustrezen DataSource*/

     /*Za shranjevanje lahko uporabimo razred TableAdapter in metodo Update
       Ta je v tem primeru bolj smiselna,saj se shranijo podatki le v točno
       določeno tabelo*/
     pisateljiTableAdapter.Update(pisateljiDataSet.Pisatelji);
}
```

Kadar naredimo spremembe v večih tabelah, pa le-te še niso shranjene v bazi, lahko s pomočjo razreda *TableAdapterManager* in njegove metode *UpdateAll* shranimo vse spremembe naenkrat.

```
/*Za shranjevanje lahko uporabimo metodo razreda TableAdapterManager in
  njegovo metodo UpdateAll. Shranijo se vsi še ne shranjeni podatki v
  vseh odprtih tabelah*/
tableAdapterManager1.UpdateAll(this.pisateljiDataSet);
```

V orodjarno *BindingNavigator* dodajmo še gumb za preklic sprememb in dodatni gumb za brisanje vrstice. Pred preklicem sprememb in pred dokončnim brisanjem želimo tudi uporabnikovo potrditev. Na desni strani orodjarne *BindingNavigator* odpremo spustni seznam

in dodamo dva gumba. Po želji za oba gumba izberemo ustrezni sliki, nato pa dvoklinimo na enega in drugega in zapišemo ustrezni odzivni metodi.

```
//odzivna metoda gumba za brisanje vrstice
private void toolStripButton2_Click(object sender, EventArgs e)
{
    //za brisanje postavke zahtevamo uporabnikovo potrditev
    if (MessageBox.Show("Brišem postavko", "BRISANJE",
MessageBoxButtons.OKCancel, MessageBoxIcon.Question) == DialogResult.OK)
    {
        int stvrstice = dataGridView1.CurrentCell.RowIndex;
        //Odstranimo vrstico iz gradnika DataGridView
        dataGridView1.Rows.RemoveAt(stvrstice);

        //Sledi še dokončno ažuriranje v bazi podatkov
        pisateljiTableAdapter.Update(pisateljiDataSet.Pisatelji);
    }
}
```

Za shranjevanje novih oz. ažuriranje spremenjenih podatkov lahko uporabljamo razreda *TableAdapter* in *TableAdapterManager*. Razlika med njima je ta, da se prvi nanaša na le na določeno tabelo v bazi podatkov, drugi pa na vse tabele hkrati.

```
//metoda za preklic (še ne ažuriranih oz. neshranjenih ) sprememb v bazi
private void preklicStripButton1_Click(object sender, EventArgs e)
{
    pisateljiDataSet.Pisatelji.RejectChanges();
    /*takole pa bi preklicali spremembe v vseh tabelah baze, seveda le v
      primeru, če je v bazi več tabel:
      pisateljiDataSet.RejectChanges();*/
}
```

DataGridView, prikaz podatkov in testiranje pravilnosti vnosa podatkov

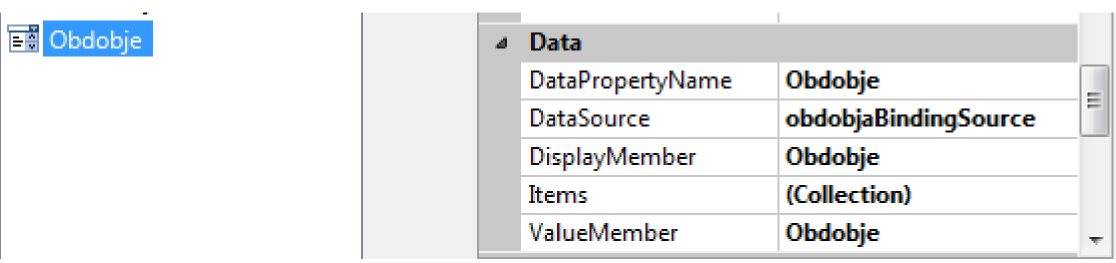
Gradnik *DataGridView*, ki smo ga povezali s tabelo v bazi podatkov, lahko v fazi načrtovanja projekta tudi poljubno oblikujemo.

Ustvarimo nov projekt *PisateljiDataGridView*. V projekt vključimo nov *DataSet* (desni klik na projekt v oknu *SolutionExplorer→Add→New Item→*izberemo *DataSet*) in ga poimenujemo *DSPisatelji*. Na levi strani razvojnega okolja se prikaže okno *DataBase Explorer*, v katerem desno kliknemo in izberemo *Add Connection*. Odpre se okno *Add Connection*, v katerem izberemo *Microsoft SQL Server Database File*. S klikom na gumb *Browse* poiščemo bazo podatkov *Pisatelji.mdb* in s klikom na gumb *OK* okno zapremo. V oknu *DataBase Explorer* kliknemo na *Pisatelji.mdf→Tables* in v okno *DSPisatelji* povlečemo tabelo *Pisatelji*. Na obrazec *Form1* sedaj postavimo gradnik *DataGridView*, mu lastnost *Dock* nastavimo na *Fill* in ga preko lastnosti *DataSource* povežemo s tabelo *Pisatelj*. V gradniku *DataGridView* se pojavijo stolpci, ki jih bomo sedaj oblikovali. Kliknemo na gumb s trikotnikom na zgornjem desnem robu gradnika *DataGridView* in izberemo *Edit Columns*, da se odpre okno *Edit Columns*. V stolpcu *Obdobje* želimo pripraviti spustni seznam, tako da bo uporabnik obdobje lahko le izbral, ne pa pisal nekaj

svojega. V oknu *Edit Columns* izberemo vrstico obdobje, poiščemo lastnost *ColumnType* in jo spremenimo v *DataGridViewComboBoxColumn*. Od tu naprej imamo na voljo dva načina:

- Poiščemo lastnost *Items* in s klikom na tripičje odpremo urejevalnik *String Colllection Editor*. Vnesemo vrstici *Moderna* in *Romantika*, ali pa še več vrstic različnih imen obdobij in urejevalnik zapremo;

- Stolpec povežemo s podatkovnim izvorom, npr. s tabelo *Obdobja*, ki pa jo moramo v ta nemen še ustvariti. Za naš primer je dovolj, da ima tabela eno samo polje z imenom Obdobje, polje naj bo ključno in tipa *nvarchar*(50).S pomočjo spustnega seznama zatem določimo lastnost *DataSource*, nato pa še lastnosti *DisplayMember* (kakšna vsebina se bo prikazovala v tem polju) in *ValueMember* (kakšno vrednost bo izbira vračala). V našem primeru sta obe nastavitvi enaki.



Slika 133: Povezovanje stolpca *Obdobje* s podatkovnim izvorom, to je s tabelo *Obdobje*.

Po želji lahko spremenimo tudi ostale lastnosti gradnika *DataGridView*. Projekt prevedemo in v stolpcu *Obdobje* bomo le-tega lahko le izbirali, ne pa vnašali neposredno.

Če kliknemo v katerokoli celico gradnika *DataGridView* lahko preko tipkovnice v polje vnesemo novo vsebino. V primeru, da v celico, ki sicer vsebuje numerične podatke, vnesemo alfanumerične znake, dobimo pri poskusu premika na drugo celico obvestilo o napaki. Okno z obvestilom v tem primeru zapremo in popravimo vsebino celice (ali pa pritisnimo tipko *Escape* za preklic vnosa). Obvestilo o napaki je privzeto, lahko pa ga nadomestimo s poljubnim lastnim obvestilom o napaki. To storimo tako, da napišemo svoj *DataError* dogodek, ki je sestavni del dogodkov gradnika *DataGridView*, npr.:

```
//odzivna metoda dogodka DataError gradnika tipa DataGridView
private void dataGridView1_DataError(object sender,
DataGridViewDataErrorEventArgs e)
{
    //testiranje napačnega vnosa datuma
    if (dataGridView1.Columns[e.ColumnIndex].DataPropertyName ==
           "DatumRojstva")
        MessageBox.Show("Napaka pri vnosu datuma!");
    //obvestilo o kakršnikoli napaki
    else MessageBox.Show("Napaka pri vnosu datuma!");
}
```

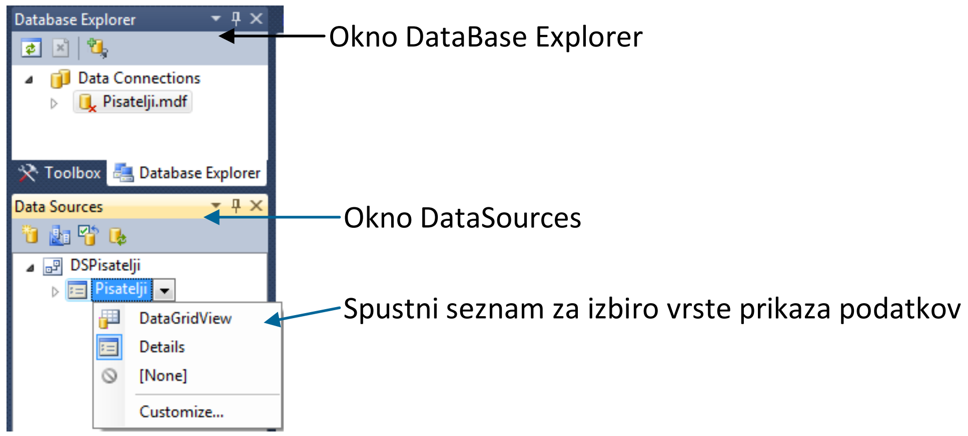
Za shranjevanje in dodajanje novih zapisov moramo seveda projekt opremiti še z gradnikom *BindingNavigator* ali pa shranjevanje spremenjenih podatkov rešiti programsko.

Prikaz podatkov iz baze v načinu *Details* – podrobnostni pregled.

Poljubna tabela baze podatkov je na obrazcu lahko prikazana v dveh oblikah

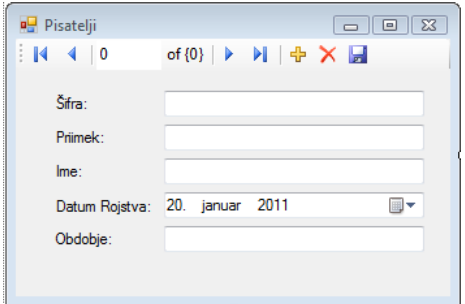
- v obliki *DataGridView*: tabelarični pogled. To je pogled, ki smo ga uporabili v vseh dosedanjih primerih;
- v obliki *Details*: podrobni pogled. To je pogled, pri katerem so na obrazcu le podatki enega samega zapisa iz tabele.

Poglejmo, kako bi namesto tabelaričnega pogleda na tabelo *Pisatelji* naredili podrobnostni pregled. Na obrazec v tem primeru ne postavimo gradnika *DataGridView*, ampak v glavnem meniju razvojnega okolja izberemo postavko *Data* in nato možnost *ShowDataSources*. Na levem robu se pokaže okno *DataSources*. Razširimo vrstico *DSPisatelji* in kliknemo na tabelo *Pisatelji*, da se prikaže spustni seznam. V seznamu izberemo možnost *Details*.



Slika 134: Okni *DataBase Explorer* in *Data Sources*, ter izbira vrste prikaza podatkov.

V oknu *DataSources* nato izberemo tabelo *Pisatelji* in jo z miško potegnemo na obrazec, da dobimo naslednjo sliko.



Slika 135: Prikaz vsebine tabele iz baze podatkov v podrobnem načinu (*Details*).

Če na obrazcu še ni bilo gradnika *BindingNavigator*, se ta pojavi sedaj. Z njegovo pomočjo se bomo premikali po tabeli. Pod navigatorjem se pojavijo gradniki za prikaz podatkov. Imena polj iz tabele *Pisatelji* so prikazana na oznakah (labelah), poleg njih pa so gradniki, ki ustrezajo tipom podatkov. Večinoma so to gradniki tipa *TextBox*, datum pa je v našem primeru prikazan v gradniku *DateTimePicker*. Ko projekt prevedemo se s pomočjo navigatorja premikamo po tabeli, posamezni podatki tekoče vrstice tabele pa so prikazani na obrazcu. Dodajamo lahko tudi nove zapise, jih brišemo in ažuriramo.

Med ustvarjanjem prejšnjega projekta je razvojno okolje v kodo obrazca dodalo dve metodi: metodo za shranjevanje podatkov v tabelo in metodo za prenos podatkov iz tabele *Pisatelji* v objekt tipa *DataSet*.

```
//Metoda za shranjevanje novih/ažuriranih podatkov v pogledu Details
private void pisateljiBindingNavigatorSaveItem_Click(object sender, EventArgs
e)
{
    this.Validate();
    this.pisateljiBindingSource.EndEdit();
    this.tableAdapterManager.UpdateAll(this.dSPisatelji);
}
/*Metoda za prenos podatkov iz baze v objekt dSPisatelji.Pisatelji ki je tipa
  DSPisatelji*/
private void Form1_Load(object sender, EventArgs e)
{
    this.pisateljiTableAdapter.Fill(this.dSPisatelji.Pisatelji);
}
```

Programska uporaba ADO.NET (brez čarovnika)

Pri izdelavi bolj kompleksnih aplikacij nam uporaba čarovnika v večini primerov ne bo zadoščala. Za dostop do baze podatkov in ustvarjanje ustrezne poizvedbe potrebujemo imenski prostor *System.Data.SqlClient*, ki ga dodamo v naš projekt.

```
using System.Data.SqlClient;
```

Imenski prostor *System.Data.SqlClient* vsebuje specializirane *ADO.NET* razrede, ki se uporabljajo za dostop do *SQL* strežnika. Med njimi je najpomembnejši razred *SQLConnection*, ki je namenjen povezavam z bazami podatkov tipa *SQL Server*.

```
SqlConnection povezava = new SqlConnection();
```

Objekt tipa *SqlConnection* lahko uporabimo na več načinov. Najpogostejši so trije:

- ustvariti želimo samo neko *poizvedbo* (npr. nek *SELECT* stavek) in podatke prikazati v enem od vizuelnih gradnikov na obrazcu;
- izvesti želimo neko obdelavo tabele v bazi in rezultate prikazati npr. v sporočilnem oknu;
- izvesti želimo pravo *SQL transakcijo* v bazi podatkov (npr. *INSERT* ali pa *UPDATE* stavek).

Tako pri poizvedbi, kot tudi pri pravi SQL transakciji je nujna uporaba varovanega bloka, saj pri povezavah lahko pride do številnih problemov in obvestilo o napaki ter vrsti napake je še kako potrebno in dobrodošlo.

Izdelava poizvedbe in prikaz podatkov v gradniku DataGridView

Vsebino tabele *Pisatelji* iz baze podatkov *Pisatelji* bi radi prikazali v gradniku *DataGridView*, ki je na obrazcu. Potrebujemo objekte tipa *SqlConnection*, *SqlDataAdapter* in *DataSet*. Njihov pomen je naslednji:

- Z objektom *SqlConnection* določimo vse potrebne podatke o izvoru podatkov. Konstruktorju posredujemo podatke o izvoru. Uporabimo *ConnectionString*, ki smo ga spoznali v datoteki *App.config*. Kadar želimo ustvariti le neko poizvedbo iz baza podatkov, objekta tipa *SqlConnection* ne potrebujemo, saj lahko povezavo in poizvedbo hkrati zagotovimo z objektom tipa *SqlDataAdapter*. Objekt tipa *SqlConnection* navadno uporabljamo v povezavi z objektom *SqlDataReader*.

- Objekt *SqlDataAdapter* zagotavlja komunikacijo med našo aplikacijo in tabelo v bazi podatkov. Njegova naloga je povezava z bazo, izvršitev ustrezne poizvedbe, shranjevanje pridobljenih podatkov v objekt tipa *DataSet*, ter vračanje ažuriranih podatkov nazaj v bazo podatkov.

- Objekt *DataSet* se uporablja za shranjevanje podatkov, ki smo jih s pomočjo objekta *SQLTableAdapter* pridobili iz neke tabele v bazi podatkov. Podatki so shranjeni v delovnem pomnilniku.

Razlika med razredoma *TableAdapter* in *SQLDataAdapter* je v tem, da je *TableAdapter* vizuelna komponenta, ki jo razvojno okolje uporablja za prikaz vseh podrobnosti povezave z bazo podatkov, SQLDataAdapter pa je razred, preko katerega vzpostavimo povezavo in hkrati tudi poizvedbo z bazo podatkov s kodiranjem.

Naslednji primer prikazuje, kako bi podatke iz tabele *Pisatelji* prikazali v tabelarični obliki programsko. Komunikacijo z bazo vzpostavimo z objektom *daPisatelji*, ki ga izpeljemo iz razreda *SqlDataAdapter*. Konstruktorju pošljemo želeno poizvedbo in objekt za povezavo z bazo. Podatke iz baze nato pridobimo tako, da ustvarimo objekt tipa *DataSet*, z metodo *Fill* objekta *daPisatelji* pa le-tega napolnimo s podatki iz tabele *Pisatelji*. Gradnik *DataGridView* sedaj s pomočjo lastnosti *DataSource* le še povežemo z objektom *dsPisatelji* in izbrano tabelo. Vsaka tabela iz baze ima v objektu tipa *DataSource* svoj indeks. Tako ima tabela *Pisatelji* v objektu *dsPisatelji* indeks 0 in do nje pridemo preko *dsPisatelji.Tables[0]*.

```
//Objekt tipa SqlDataAdapter za komunikacijo s tabelo v bazi podatkov
SqlDataAdapter daPisatelji;
//Objekt tipa DataSet za shranjevanje pridobljenih podatkov iz baze
DataSet dsPisatelji;

//Pri delu s poizvedbami navadno uporabimo varovalni blok
try
{
```

```
    //Povezovalni niz ConnectionString z bazo podatkov
    string connectionString = @"Data 
Source=.\SQLEXPRESS;AttachDbFilename=C:\DOKUMENTI\APJ+ŠOLA\APJ\SQL 
SERVER\Pisatelji\Pisatelji.mdf;Integrated Security=True;Connect 
Timeout=30;User Instance=True";

    string poizvedba = "SELECT * FROM Pisatelji";//Vsebina poizvedbe
    //Objekt tipa SqlAdapter vzpostavi komunikacijo z bazo
    daPisatelji = new SqlDataAdapter(poizvedba, connectionString);
    //Objekt za shranjevanje pridobljenih podatkov v pomnilniku
    dsPisatelji = new DataSet();
    //Podatke iz baze z metodo Fill dejansko prenesemo v ustrezen DataSet
    daPisatelji.Fill(dsPisatelji, "Pisatelji");
    //Gradnik DataGridView povežemo s podatki gradnika DataSet v pomnilniku
    //do tabel v bazi dostopamo preko indeksa (Tables[0] je prva tabela)
    dataGridView1.DataSource = dsPisatelji.Tables[0];
}
catch
{
    MessageBox.Show("Napaka pri dostopu do baze podatkov!");
}
```

Pogosto želimo v gradniku *DataGridView* prikazati le določene podatke iz neke tabele. Naslednji primer prikazuje, kako lahko iz tabele *Pisatelji* prikažemo le določene pisatelje.

Obdelava tabele v bazi podatkov

Ugotoviti želimo število pisateljev iz tabele *Pisatelji*, ki so rojeni po letu 1900. V ta namen bomo tabelo *Pisatelji* le obdelali. Prebrali bomo vse vrstice iz tabele, eno za drugo. Uporabili bomo objekt razreda *SqlDataReader*, ki ima podobno vlogo kot objekta razreda *StreamReader* pri branju vrstic iz tekstovne datoteke. Razred *SqlDataReader* je bil narejen z namenom, da podatke iz tabel pridobiva enega za drugim in ne pušča zaklenjenih vrstic v tabeli, ko je vsebina vrstice enkrat pridobljena. Tak način pridobivanja podatkov (vrstic) iz tabel seveda pomeni veliko prednost pri izboljšanju konkurenčnosti naše aplikacije. Obenem je uporaba objektov izpeljanih iz razreda *SqlDataReader* tudi najhitrejši način pridobivanja podatkov iz baze.

Za branje celotnega zapisa iz določene tabele bomo zato ustvarili objekt *dataReader* tipa *SqlDataReader* in uporabili njegovo metodo *Read()*. Metoda vrne *False*, če podatki ne obstajajo, ali pa smo na koncu tabele. Do posameznih polj prebranega zapisa pridemo z metodo *GetValue* objekta *dataReader* preko indeksov.

```
//Povezovalni niz z bazo podatkov
string connectionString = @"Data 
Source=.\SQLEXPRESS;AttachDbFilename=C:\DOKUMENTI\APJ+ŠOLA\APJ\SQL 
SERVER\Pisatelji\Pisatelji.mdf;Integrated Security=True;Connect 
Timeout=30;User Instance=True";

SqlConnection povezava;//Objekt za povezavo
povezava = new SqlConnection(connectionString);

//Povezavo z bazo lahko ustvarimo na več načinov. Prikazana sta dva od njih!
```

```
//1. NAČIN
//SqlCommand poizvedba = new SqlCommand();
//poizvedba.CommandText = "SELECT * FROM Pisatelji";
//poizvedba.Connection = povezava;
```

```
//2.NAČIN
//Vrsto interakcije z bazo napovemo z objektom tipa SqlCommand
SqlCommand poizvedba = new SqlCommand("SELECT * FROM Pisatelji", povezava);
//Odpremo povezavo z bazo
poizvedba.Connection.Open();
/*Ustvarimo podatkovni tok za branje vrstic iz tabele v bazi. Najhitrejši
  način za pridobivanje podatkov iz SQL Server baze podatkov je z uporabo
  razreda SqlDataReader. Ta razred pridobi vrstice iz baze podatkov
  najhitreje kot omrežje sploh omogoča in podatke preda naši aplikaciji*/
SqlDataReader dataReader = poizvedba.ExecuteReader();
int skupaj = 0; //začetno število pisateljev, rojenih po letu 1900

//metoda Read vrne true, če je poizvedba uspešna, sicer vrne false.
while (dataReader.Read())//dokler obstajajo podatki
{
    /*ker smo v SELECT stavku navedli vse podatke iz tabele Pisatelji ima
      podatek o datumu rojstva znotraj enega stavka indeks enak 3!*/
    if (Convert.ToDateTime(dataReader.GetValue(3)).Year >1900)
            skupaj++;
}
dataReader.Close();//zapremo podatkovni tok
povezava.Close();//Zapremo povezavo z bazo

//Rezultat obdelave prikažemo v sporočilnem oknu
MessageBox.Show("Pisateljev rojenih po letu 1900: "+skupaj);
```

Uporabili smo tudi razred *SqlCommand*. Z objektom tega tipa povemo, kakšno vrsto interakcije z bazo želimo. Osnovne možnosti so *select*, *insert*, modify in delete. V zgornjem primeru smo uporabili le poizvedbo s stavkom *select*. Dejansko pa smo podatke iz tabele nato pridobili s pomočjo objekta tipa *SqlDataReader*. http://www.connectionstrings.com/

Datoteka App.Config

Zapis poti do mape, v kateri je baza podatkov, kar znotraj programa (v nizu *connectionString*) je za potrebe vaj in testiranja aplikacije sicer v redu, a s stališča programiranja nepraktična. Mapa z bazo podatkov se lahko spremeni in v tem primeru je potrebno popravljanje programa, ponovno prevajanje in nameščanje na strežnik. Zaradi tega povezovalni niz z bazo podatkov zapišemo v datoteko *App.Config*, ki jo najdemo v oknu *Solution Explorer*. Datoteko odpremo in za stavkom *<configuration>* zapišemo ustrezne stavke npr. takole:

```
<connectionStrings>
   <add name="dBKnjige"
          connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=C:\
WPBazeProgramsko\KnjigeSQL.mdf;Integrated Security=true;User Instance=True"
          providerName="System.Data.SqlClient" />
```

```
</connectionStrings>
```

Spremembe v konfiguracijski datoteki *App.Config* shranimo in datoteko zapremo. Vrnemo se v spletni obrazec, v katerem smo prej zapisali povezavo in poizvedbo z bazo podatkov in najprej pobrišemo stavek v katerem smo določili povezovalni niz (stavek *string connectionString=…*). V *using* sekciji našega obrazca najprej dodamo imenski prostor *System.Configuration*.

```
using System.Configuration;
```

S pomočjo tega imenskega prostora moramo sedaj iz datoteke Web.Config pridobiti povezovalni niz (connectionString). To storimo s stavkom

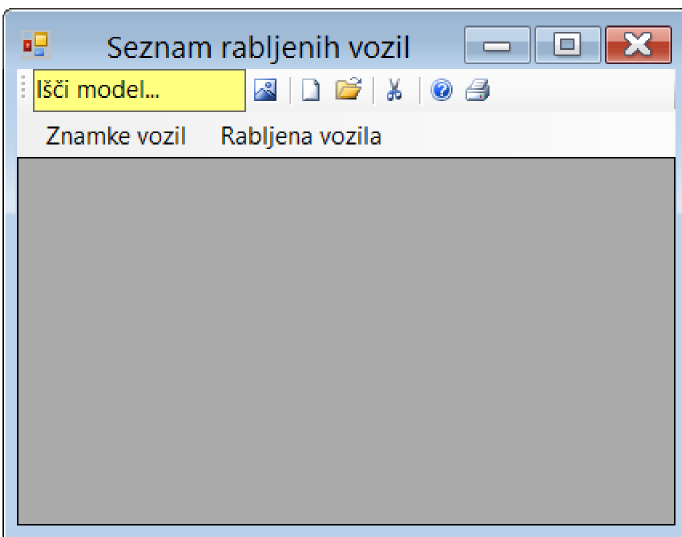
```
string connectionString=
ConfigurationManager.ConnectionStrings["dBKnjige"].ConnectionString;
```

Kaj smo s tem pridobili: ča se bo kadarkoli spremenilo ime strežnika, mapa, ali pa celotna pot do naše baze podatkov, bo potrebno le spremeniti vsebino vrstice connectionString v datoteki App.Config. To pa je navadna tekstovna datoteka in ni potreben poseg v projekt.

Zbirka RabljenaVozila

Na strežniku http://uranic.tsckr.si/Lokalne%20baze/ je tudi arhivirana zbirka *RabljenaVozila.mdf*. Vsebuje dve tabeli in v nasLednji vaji bomo prikazali kompletno manipulacijo z obema tabelama.

Glavni obrazec projekta vsebuje orodjarno, vrstico z meniji, osrednji del pa zavzema gradnik *DataGridView*, v katerem želimo prikazati tabelo *Avtomobili*.



Slika 136: Glavni obrazec projekta *RabljenaVozila*

V orodjarno smo dodali vnosno polje za iskenje določenega modela v gradniku *DataGridView*, ter gumbe, ki imajo naslednjo vlogo(od leve proti desni): ustrajanje poizvedbe, dodajanje, ažuriranje, brisanje, pomoč in tiskanje. V konstruktorju tega obrazca poskrbimo za prikaz vsebine tabele *Avtomobili*:

```
SqlDataAdapter daAvtomobili;
DataSet dsRabljenaVozila;
string connectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\RabljenaVozila.mdf;Integrate
d Security=True;User Instance=True";
public FRabljenaVozila()
{
InitializeComponent();
try
{
    /*tule ne bomo uporabili using stavka, ker želimo ohraniti povezavo z izvorom
podatkov: zaradi ažuriranja gradnika dataGridView1*/
    //poizvedba: zapišemo ustrezen pogoj
    string poizvedba = "SELECT * FROM Avtomobili";
    daAvtomobili = new SqlDataAdapter(poizvedba, connectionString);
    dsRabljenaVozila = new DataSet();
    daAvtomobili.Fill(dsRabljenaVozila, "Avtomobili");
    dataGridView1.DataSource = dsRabljenaVozila.Tables[0];

    dataGridView1.Columns[0].Visible = false;//prvi stolpec bo skrit
    dataGridView1.Columns["Znamka"].Width = 140;//širina stolpce 'Znamka'
    dataGridView1.Columns["Znamka"].Width = 90;//širina stolpce 'Znamka'
    dataGridView1.Columns["Letnik"].Width = 55;//širina stolpce 'Letnik'
}
catch
{
    MessageBox.Show("Napaka pri dostopu do baze podatkov!");
}
```

Za prikaz nismo uporabili *using* stavka, ker bomo objekt *daAvtomobili* potrebovali še kasneje, ob vsakem ažuriranju tabele *Avtomobili*. Gumbu za dodajanje novega vozila v tabelo priredimo naslednje stavke:

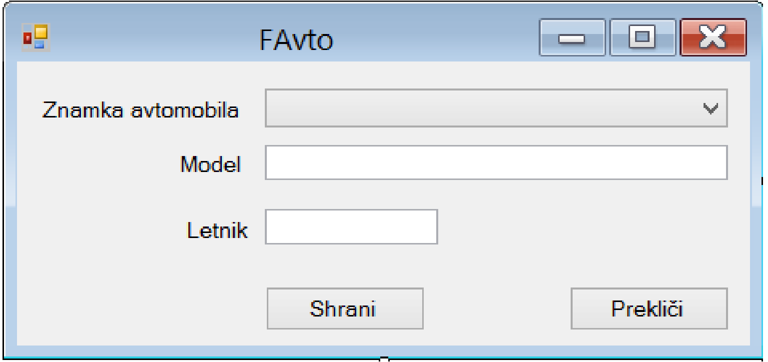
```
FAvto Nov = new FAvto(); //nov obrazec za vnos avtomobila
Nov.Text = "Vnos novega avtomobila!";
//Ko se obrazec odpre, bo v spustnem seznamu že neka vrednost
Nov.comboBox1.SelectedIndex = 0;
if (Nov.ShowDialog() == DialogResult.OK)
{
    try
    {
        using (SqlConnection dataConnection = new
SqlConnection(connectionString))
        {
            dataConnection.Open();
            string poizvedba = "Insert INTO Avtomobili (Znamka,Model,Letnik)
VALUES (@Znamka,@Model,@Letnik)";
            using (SqlCommand dataCommand = new SqlCommand(poizvedba,
dataConnection))
```

```
            {
                //deklaracija parametrov za SQL transacijo
                dataCommand.Parameters.Add(new SqlParameter("@Znamka",
Nov.comboBox1.Text));
                //LAHKO TUDI TAKOLE:
dataCommand.Parameters.AddWithValue("@znamkaVozila", znamkaVozila);
                dataCommand.Parameters.Add(new SqlParameter("@Model",
Nov.textBox1.Text));
                dataCommand.Parameters.Add(new SqlParameter("@Letnik",
Nov.textBox2.Text));
                dataCommand.ExecuteNonQuery();
            }
        }
        //dosedanje podatke v objektu DataSet pobrišemo
        dsRabljenaVozila.Clear();
        //osvežimo podatke v objektu DataSet
        daAvtomobili.Fill(dsRabljenaVozila, "Avtomobili");
        //postavimo se na zadnji zapis v tabeli
        int pozicija = dataGridView1.Rows.Count - 1;
        //v gradniku DataGridView se postavimo povsem na konec
        dataGridViewIzbran(pozicija);
    }
    catch
    {
        MessageBox.Show("Napaka!");
    }
}
```

V kodi smo uporabili tudi klic metode dataGridViewIzbran, ki poskrbi za to, da je na ekranu že izbrano vozilo, ki smo ga pravkar vnesli.

```
//metoda za določanje izbranega zapisa v gradniku DataGridView
private void dataGridViewIzbran(int vrstica)
{
    try
    {
        if (vrstica == -1)
            vrstica = 0;
        dataGridView1.ClearSelection();//doslej izbrane zapise najprej odznačimo
        dataGridView1.FirstDisplayedScrollingRowIndex = vrstica;/*vsebino
gradnika DataGridView premaknemo tako, da bo na vrhu izbrani zapis*/
        dataGridView1.Rows[vrstica].Selected = true;
        dataGridView1.CurrentCell = dataGridView1.Rows[vrstica].Cells[1];
    }
    catch
    {
        dataGridView1.Rows[1].Selected = true;
        dataGridView1.CurrentCell = dataGridView1.Rows[1].Cells[1];
    }
}
```

Obrazec FAvto, ki smo ga pripravili za vnos novega avtomobila vsebuje spustni seznam, dve vnosni polji, ter dva gumba. V spustnem seznamu želimo prikazati vsebino tabele Znamke.



Slika 137: Obrazec za vnos novega avtomobila in hkrati za ažuriranje podatkov o avtomobilu.

V kodi obrazca *FAvto* moramo poskrbeti za povezavo spustnega seznama S tabelo *Znamke*, ter za ustrezne validacije vnosov, pred shranjevanjem. Tule je ustrezna koda:

```
//SqlDataAdapter daZnamke;
DataSet dsRabljenaVozila;
string connectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\RabljenaVozila.mdf;Integrate
d Security=True;User Instance=True";
bool kontrolaModela = false, kontrolaLetnika=false;
public FAvto()
{
    InitializeComponent();
    try
    {
        string poizvedba = "SELECT * FROM Znamke";
        using (SqlDataAdapter daZnamke = new SqlDataAdapter(poizvedba,
connectionString))
        {
            dsRabljenaVozila = new DataSet();
            daZnamke.Fill(dsRabljenaVozila, "Znamke");
            comboBox1.DropDownStyle = ComboBoxStyle.DropDownList;
            comboBox1.DataSource = dsRabljenaVozila.Tables[0];
            comboBox1.DisplayMember = "Znamka";
            comboBox1.ValueMember = "Znamka";
            comboBox1.Text = comboBox1.Items[0].ToString();
        }
    }
    catch
    {
        MessageBox.Show("Napaka pri dostopu do baze podatkov!");
    }
}

private void textBox2_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((e.KeyChar < '0' || e.KeyChar > '9') && e.KeyChar != (char)8)
        e.Handled = true;  //dovoljen le vnos cifer in brisanje vnesenih znakov
}

private void textBox1_Validating(object sender, CancelEventArgs e)
{
```

```
    if (textBox1.Text.Length == 0)
    {
        errorProvider1.SetError(textBox1, "Model avtomobila ni vnesen!");
        kontrolaModela= false;
    }
    else
    {
        errorProvider1.SetError(textBox1, "");
        kontrolaModela = true;
    }
}

int letnik = 0;
private void textBox2_Validating(object sender, CancelEventArgs e)
{
    try
    {
        int letnik = Convert.ToInt32(textBox2.Text);
        errorProvider1.SetError(textBox2, "");
        kontrolaLetnika = true;
    }
    catch
    {
        errorProvider1.SetError(textBox2, "Napaka pri vnosu letnika");
        kontrolaLetnika = false;
    }
}

private void button1_Click(object sender, EventArgs e)
{   /*validacija obeh vnosnih polje je potrebna zato, ker se isti obrazec
      uporablja tudi za urejanje podatkov!!!*/
    textBox1_Validating(textBox1, null);
    textBox2_Validating(textBox2, null);
    if (kontrolaModela && kontrolaLetnika)
        this.DialogResult = DialogResult.OK;
    else this.DialogResult = DialogResult.None;
}
```

Potrebujemo še odzivno metodo gumba za dodajanje novega vozila v glavnem obrazcu:

```
private void dodajNovoVoziloToolStripMenuItem_Click(object sender, EventArgs e)
{
    FAvto Nov = new FAvto();
    Nov.Text = "Vnos novega avtomobila!";
    Nov.comboBox1.SelectedIndex = 0;//Ko se obrazec odpre, bo v spustnem seznamu
že neka vrednost
    if (Nov.ShowDialog() == DialogResult.OK)
    {
        try
        {
            using (SqlConnection dataConnection = new
SqlConnection(connectionString))
            {
                dataConnection.Open();
                string poizvedba = "Insert INTO Avtomobili (Znamka,Model,Letnik)
VALUES (@Znamka,@Model,@Letnik)";
```

```
                using (SqlCommand dataCommand = new SqlCommand(poizvedba,
dataConnection))
                {
                    //deklaracija parametrov za SQL transacijo
                    dataCommand.Parameters.Add(new SqlParameter("@Znamka",
Nov.comboBox1.Text));
                    /*LAHKO TUDI TAKOLE:
dataCommand.Parameters.AddWithValue("@znamkaVozila", znamkaVozila);*/
                    dataCommand.Parameters.Add(new SqlParameter("@Model",
Nov.textBox1.Text));
                    dataCommand.Parameters.Add(new SqlParameter("@Letnik",
Nov.textBox2.Text));
                    dataCommand.ExecuteNonQuery();
                }
            }
            //dosedanje podatke v objektu DataSet pobrišemo
            dsRabljenaVozila.Clear();
            //osvežimo podatke v objektu DataSet
            daAvtomobili.Fill(dsRabljenaVozila, "Avtomobili");
            //postavimo se na zadnji zapis v tabeli
            int pozicija = dataGridView1.Rows.Count - 1;
            //v gradniku DataGridView se postavimo povsem na konec
            dataGridViewIzbran(pozicija);
        }
        catch
        {
            MessageBox.Show("Napaka!");
        }
    }
}
```

Obrazec *FAvto* uporabimo tudi za ažuriranje podatka o poljubnem vozilu. Potrebujemo pa odzivno metodo gumba za urejanje na glavnem obrazcu, ki je obenem tudi odzivna metoda dogodka *DoubleClick* gradnika *DataGridView*.

```
private void dataGridView1_DoubleClick(object sender, EventArgs e)
{
    FAvto Uredi = new FAvto();
    Uredi.Text = "Urejanje podatkov o avtomobilu!";
    int izbran = Convert.ToInt32(dataGridView1.CurrentRow.Cells[0].Value);
    int pozicija = dataGridView1.CurrentRow.Index;

    Uredi.comboBox1.Text = dataGridView1.CurrentRow.Cells[1].Value.ToString();
    Uredi.textBox1.Text = dataGridView1.CurrentRow.Cells[2].Value.ToString();
    Uredi.textBox2.Text = dataGridView1.CurrentRow.Cells[3].Value.ToString();
    if (Uredi.ShowDialog() == DialogResult.OK)
    {
        try
        {
            using (SqlConnection dataConnection = new
SqlConnection(connectionString))
            {
                dataConnection.Open();
                string poizvedba = "Update Avtomobili  SET
Znamka=@Znamka,Model=@Model,Letnik=@Letnik  WHERE Zaporedna=@Zaporedna";
```

```
                using (SqlCommand dataCommand = new SqlCommand(poizvedba,
dataConnection))
                {
                    //deklaracija parametrov za SQL transacijo
                    dataCommand.Parameters.Add(new SqlParameter("@Zaporedna",
izbran));
                    dataCommand.Parameters.Add(new SqlParameter("@Znamka",
Uredi.comboBox1.Text));
                    dataCommand.Parameters.Add(new SqlParameter("@Model",
Uredi.textBox1.Text));
                    dataCommand.Parameters.Add(new SqlParameter("@Letnik",
Uredi.textBox2.Text));
                    dataCommand.ExecuteNonQuery();
                }
            }
        }
        catch
        {
            MessageBox.Show("Šifra skupine že obstaja ali pa napaka pri
shranjevanju!");
        }
        //potrebno je še ažurirati stanje v gradniku DataGridView
        dsRabljenaVozila.Clear();
        daAvtomobili.Fill(dsRabljenaVozila, "Avtomobili");
        //postavimo se na isti zapis v tabeli
        dataGridViewIzbran(pozicija);
    }
}
```

Ker pa se uporabnik pri vnosih lahko zmoti, mu omogočimo še brisanje poljubnega vozila iz tabele. Za to bo poskrbela odzivna metoda gumba za brisanje na glavnem obrazcu.

```
private void brišiIzbranoVoziloToolStripMenuItem_Click(object sender, EventArgs
e)
{
    if (MessageBox.Show("Brišem vrstico " +
dataGridView1.CurrentRow.Cells[1].Value.ToString(), "Brisanje vrstice",
MessageBoxButtons.OKCancel, MessageBoxIcon.Question) == DialogResult.OK)
    {
        //shranim pozicijo vrstice, ki jo brišem
        int pozicija = dataGridView1.CurrentRow.Index;
        try
        {
            int zaporedna =
Convert.ToInt32(dataGridView1.CurrentRow.Cells[0].Value);
            using (SqlConnection dataConnection = new
SqlConnection(connectionString))
            {
                dataConnection.Open();
                string poizvedba = "DELETE FROM Avtomobili WHERE Zaporedna
=@Zaporedna";
                using (SqlCommand dataCommand = new SqlCommand(poizvedba,
dataConnection))
                {
```

```
                    dataCommand.Parameters.Add(new SqlParameter("@Zaporedna",
zaporedna));
                    dataCommand.CommandType = CommandType.Text;
                    dataCommand.ExecuteNonQuery();
                }
            }
        }
        catch
        {
            MessageBox.Show("Napaka pri brisanju zapisa!");
        }
        /*Osvežimo podatke v gradniku DataGridView glede na novo stanje baze
          podatkov */
        dsRabljenaVozila.Clear();
        daAvtomobili.Fill(dsRabljenaVozila, "Avtomobili");
        dataGridViewIzbran(pozicija);
    }
}
```

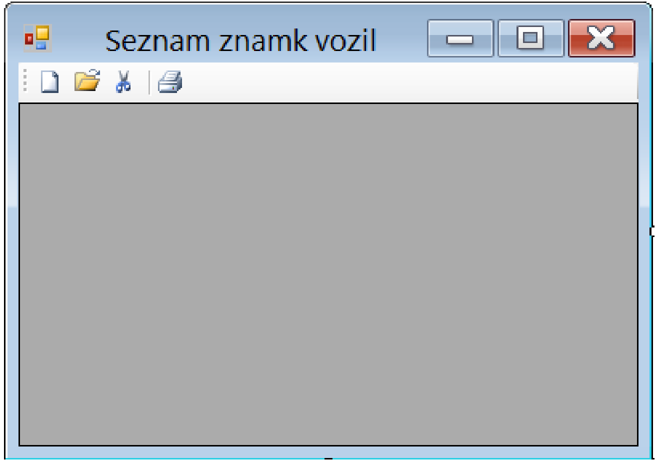
Tule sta še odzivni metodi za iskanje določenega modela vozila (iskanje v drugem stolpcu), ter za enostavno tiskanje seznama vozil.

```
//odzivna metoda za iskanje modela vozila
private void toolStripButton1_Click(object sender, EventArgs e)
{
    int pozicija = -1;
    for (int i = 0; i < dataGridView1.Rows.Count; i++)
    {
        if (Convert.ToString(dataGridView1.Rows[i].Cells[2].Value) ==
toolStripTextBox1.Text)
        {
            pozicija = i;
            break;
        }
    }
    if (pozicija == -1)
        MessageBox.Show("Ta model ne obstaja!");
    else
    {
        dataGridViewIzbran(pozicija);
    }
}
int stVrstice = 0;
/*odzivna metoda gumba za tiskanje: na obrazcu potrebujemo še gradnik tipa
PrintDocument*/
private void printToolStripButton_Click(object sender, EventArgs e)
{
    if (printDialog1.ShowDialog() == DialogResult.OK)
    {
        printDocument1.PrinterSettings = printDialog1.PrinterSettings;
        stVrstice = 0;
        printPreviewDialog1.Document = printDocument1;
        printPreviewDialog1.ShowDialog();
    }
}
```

```
Font font =new Font("Calibri", 11, FontStyle.Regular);
//odzivna metoda PrintPage objekta printDocument1
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 strani
    float zgornjiRob = e.MarginBounds.Top;//oddaljenost od vrha strani
    float yPos = zgornjiRob + font.GetHeight(e.Graphics); //razdalja od zgornjega
roba, kjer bomo pričeli s tiskanjem
    //natisnemo glavo strani
    e.Graphics.DrawString("Znamka, model in letnik vozila", new Font("Calibri",
12, FontStyle.Bold), Brushes.Black, leviRob, yPos, new StringFormat());
    yPos = yPos + 2 * font.GetHeight(e.Graphics);
    e.Graphics.DrawLine(new Pen(Color.Black, 1), leviRob,yPos, leviRob+300,yPos);
    yPos = yPos + 2 * font.GetHeight(e.Graphics);
    int natisnjenihVrstic=0;//število že natisnjenih vrstic na eni strani
    /*tiskajmo še posamezne podatke (vrstice), dokler jih ne zmanjka: tiskali
bomo 30 vrstic na stran*/
    while (natisnjenihVrstic<30 && stVrstice < dataGridView1.Rows.Count)
    {
        //oblikujemo posamezno vrstico
        string vrstica = dataGridView1.Rows[stVrstice].Cells[1].Value.ToString()
+ ",     " + dataGridView1.Rows[stVrstice].Cells[2].Value.ToString() + ",    " +
dataGridView1.Rows[stVrstice].Cells[3].Value.ToString();
        //in jo natisnemo
        e.Graphics.DrawString(vrstica, font, Brushes.Black, leviRob, yPos, new
StringFormat());
        stVrstice++; //povečamo število vseh vrstic
        natisnjenihVrstic++; //povečamo število natisnjenih vrstic na tej strani
        yPos = yPos + 1.5F * font.GetHeight(e.Graphics);
    }
    if (dataGridView1.Rows.Count > stVrstice)//preverimo, če je tiskanje končano
       //podatkov še ni konec, nadaljujejo se na naslednji strani
       e.HasMorePages = true;
    else
        e.HasMorePages = false; //konec tiskanja
}
```

Ostane nam še koda za prikaz znam vozil, ki so zapisane v tabeli *Znamke*. V ta namen pripravimo obrazec *FZnamke*. Na obrazec postavimo orodjarno in v njej gradnike za dodajanje, ažuriranje, brisanje in tiskanje, dodajmo pa še gradnik *DataGridView*. Obrazec bomo odprli iz glavnega obrazca, npr. ob kliku na možnost *Znamke Vozil* menija.

```
private void pregledToolStripMenuItem_Click(object sender, EventArgs e)
{
    FZnamke Pregled = new FZnamke();
    //obrazec odpremo kot dialog, da ga uporabnik ne bi slučajno odprl dvakrat
    Pregled.ShowDialog();
}
```



Slika 138: Obrazec za prikaz tabele Znamke.

Tule je celotna koda obrazca *FZnamke*, ki vsebuje potrebne metode za ažuriranje te tabele. Za potrebe dodajanja in ažuriranja določene znamke, bomo potrebovali še obrazec *FZnamka*.

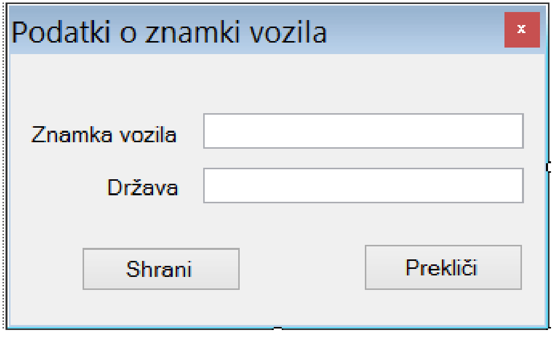
```
public partial class FZnamke : Form
{
    SqlDataAdapter daZnamke;
    DataSet dsRabljenaVozila;
    string connectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\RabljenaVozila.mdf;Integrate
d Security=True;User Instance=True";
    SqlConnection dataConnection;

    public FZnamke()
    {
        InitializeComponent();
        try
        {
            string poizvedba = "SELECT * FROM Znamke";
            daZnamke = new SqlDataAdapter(poizvedba, connectionString);
            dsRabljenaVozila = new DataSet();
            daZnamke.Fill(dsRabljenaVozila, "Znamke");
            dataGridView1.DataSource = dsRabljenaVozila.Tables[0];
            dataGridView1.Columns["Znamka"].Width = 140;//širina stolpce 'Znamka'
            dataGridView1.Columns["Drzava"].Width = 90;//širina stolpce 'Država'
        }
        catch
        {
            MessageBox.Show("Napaka pri dostopu do baze podatkov!");
        }
}

//odzivna metoda za dodajanje nove znamke avtomobila
private void newToolStripButton_Click(object sender, EventArgs e)
{
    FZnamka Nova = new FZnamka();
    Nova.Text = "Dodajanje nove znamke avtomobila";
    if (Nova.ShowDialog() == DialogResult.OK)/*uporabnik je obrazec zaprl s
klikom na gumb Shrani*/
```

```
    {
        try
        {
            using (SqlConnection dataConnection = new
SqlConnection(connectionString))
            {
                dataConnection.Open();
                string poizvedba = "Insert INTO Znamke (Znamka,Drzava) VALUES
(@Znamka,@Drzava)";
                using (SqlCommand dataCommand = new SqlCommand(poizvedba,
dataConnection))
                {
                    //deklaracija parametrov za SQL poizvedbo
                    dataCommand.Parameters.Add(new SqlParameter("@Znamka",
Nova.textBox1.Text));
                    dataCommand.Parameters.Add(new SqlParameter("@Drzava",
Nova.textBox2.Text));
                    dataCommand.CommandType = CommandType.Text;
                    dataCommand.ExecuteNonQuery();
                }
            }
        }
        catch
        {
            MessageBox.Show("Znamka vozila že obstaja ali pa napaka pri
shranjevanju!");
        }
        dsRabljenaVozila.Clear();
        daZnamke.Fill(dsRabljenaVozila, "Znamke");
        //postavimo se na zadnji zapis v tabeli
        int pozicija = dataGridView1.Rows.Count-1;
        dataGridViewIzbran(pozicija);
    }
}

//odzivna metoda za urejanje izbrane znamke avtomobila
private void openToolStripButton_Click(object sender, EventArgs e)
{
    FZnamka Uredi = new FZnamka();
    string znamkaVozila =
Convert.ToString(dataGridView1.CurrentRow.Cells[0].Value);
    string staraDrzava =
Convert.ToString(dataGridView1.CurrentRow.Cells[1].Value);
    Uredi.textBox1.Text = znamkaVozila;
    Uredi.textBox1.ReadOnly = true;//znamke ne moremo urejati- ključno polje
    Uredi.textBox2.Text = dataGridView1.CurrentRow.Cells[1].Value.ToString();
    Uredi.Text = "Urejanje znamke avtomobila";
    int pozicija = dataGridView1.CurrentRow.Index;
    if (Uredi.ShowDialog() == DialogResult.OK)/*uporabnik je obrazec zaprl s
klikom na gumb Shrani*/
    {
        try
        {
            using (SqlConnection dataConnection = new
SqlConnection(connectionString))
            {
```

```
                dataConnection.Open();
                string poizvedba = "UPDATE Znamke SET Drzava = @Drzava WHERE
Znamka = @Znamka";
                using (SqlCommand dataCommand = new SqlCommand(poizvedba,
dataConnection))
                {
                    //deklaracija parametrov za SQL poizvedbo
                    dataCommand.Parameters.Add(new SqlParameter("@Znamka",
znamkaVozila));
                    dataCommand.Parameters.Add(new SqlParameter("@StaraDrzava",
staraDrzava));
                    dataCommand.Parameters.Add(new SqlParameter("@Drzava",
Uredi.textBox2.Text));
                    dataCommand.CommandType = CommandType.Text;
                    dataCommand.ExecuteNonQuery();
                }
            }
        }
        catch
        {
            MessageBox.Show("Znamka vozila že obstaja ali pa napaka pri
shranjevanju!");
        }
        dsRabljenaVozila.Clear();
        daZnamke.Fill(dsRabljenaVozila, "Znamke");
        //postavimo se na zadnji zapis v tabeli
        dataGridViewIzbran(pozicija);
    }
}

//odzivna metoda za določanje izbranega zapisa v gradniku DataGridView
private void dataGridViewIzbran(int vrstica)
{
    try
    {
        if (vrstica == -1)
            vrstica = 0;
        dataGridView1.ClearSelection();//doslej izbrane zapise najprej odznačimo
        dataGridView1.FirstDisplayedScrollingRowIndex = vrstica;/*vsebino
gradnika DataGridView premaknemo tako, da bo na vrhu izbrani zapis*/
        dataGridView1.Rows[vrstica].Selected = true;
        dataGridView1.CurrentCell = dataGridView1.Rows[vrstica].Cells[1];
    }
    catch {  }
}

//odzivna metoda za brisanje izbrane znamke avtomobiula
private void cutToolStripButton_Click(object sender, EventArgs e)
{
    if (MessageBox.Show("Brišem vrstico " +
dataGridView1.CurrentRow.Cells[1].Value.ToString(), "Brisanje vrstice",
MessageBoxButtons.OKCancel, MessageBoxIcon.Question) == DialogResult.OK)
    {
        int pozicija = dataGridView1.CurrentRow.Index;//shranim pozicijo vrstice,
ki jo brišem
        try
```

```
        {
            string znamka = dataGridView1.CurrentRow.Cells[0].Value.ToString();
            using (SqlConnection dataConnection = new
SqlConnection(connectionString))
            {
                dataConnection.Open();
                string poizvedba = "DELETE FROM Znamke WHERE Znamka =@Znamka";
                using (SqlCommand dataCommand = new SqlCommand(poizvedba,
dataConnection))
                {
                    dataCommand.Parameters.Add(new SqlParameter("@Znamka",
znamka));
                    dataCommand.CommandType = CommandType.Text;
                    dataCommand.ExecuteNonQuery();
                }
            }
        }
        catch
        {
            //Če je prišlo do napake, vzpostavimo prejšnje stanje
            MessageBox.Show("Napaka pri brisanju zapisa!");
        }
        /*Osvežimo podatke v gradniku DataGridView glede na novo stanje baze
          podatkov*/
        dsRabljenaVozila.Clear();
        daZnamke.Fill(dsRabljenaVozila, "Znamke");
        /*postavim se nazaj na številko vrstice, ki je bila odstanjena. Če je
         bila slučajno odstranjena zadnja vrstica pa bo v prikazu izbrana prva*/
        dataGridViewIzbran(pozicija);
    }
}
```

Obrazec *FZnamka* za vnos oz. ažuriranje znamke vozila vsebuje le dve vnosni polji in gumba.



Slika 139: Obrazec za vnos oz. ažuriranje znamke vozila.

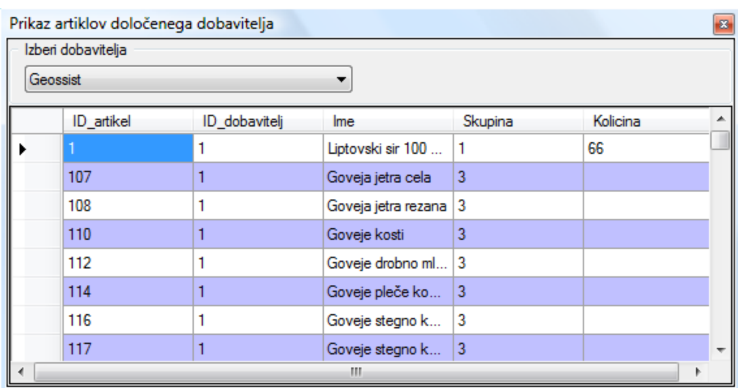
V brazcu FZnamka moramo zapisati le odzivno metodo gumba Shrani. Gumbu Prekliči smo le nastavili lastnost DialogResult na false..

```
private void Shrani_Click(object sender, EventArgs e)
{
    if (textBox1.Text.Length > 0 && textBox2.Text.Length > 0)
    {
        this.DialogResult = DialogResult.OK;
```

```
    }
    else
    {
        MessageBox.Show("Vnos znamke vozila in države je obvezen!");
        this.DialogResult = DialogResult.None;  //obrazec se ne bo zaprl
    }
}
```

Povezava med dvema tabelama: prikaz le določenih podatkov iz tabele

Pogosto želimo na obrazcu prikazati le določene podatke (zapise) iz tabele. Naslednja primera prikazujeta, kako lahko to naredimo povsem programsko in kako s pomočjo gradnika *DataSet*. Spustni seznam bomo povezali s tabelo *Dobavitelji* v bazi *NabavaSQL*, vse artikle izbranega dobavitelja iz tabele *Artikli*, pa bomo prikazali v gradniku *DataGridView* na istem obrazcu. V obeh primerih najprej na obrazec postavimo trie gradnike: gradnik *GroupBox* z napisom *Izberi Dobavitelja*, vanj postavimo gradnik *ComboBox*, preostali del obrazca pa zavzema gradnik *DataGridView*.



Slika 140: Povezava gradnika *ComboBox* s tabelo *Dobavitelji* in *DataGridView* s tabelo *Artikli*.

Programska povezava med dvema tabelama

Oba gradnika povežimo s podatkovnim izvorom najprej programsko. Povezavo zapišemo v odzivno metodo dogodka *Load* obrazca. Ustvarimo še objekta *daDobavitelji* tipa *SqlDataAdapter* in *dsNabavaSQL* tipa *DataSet*. Podatke iz tabele *Dobavitelji* lahko sedaj v objekt *dsDobavitelji* prenesemo z metodo *Fill* objekta *daDobavitelji*. Objektu tipa *ComboBox* moramo nato določiti izvor podatkov *DataSource* in še lastnosti *DisplayMember* in *ValueMember*. Pri določanju izvora bomo uporabili stavek

```
comboBox1.DataSource = dsNabavaSQL.Tables[0];
```

V objektu *dsNabavaSQL* je namreč v splošnem lahko več tabel iz baze *NabavaSQL*, mi pa želimo povezavo z eno samo. Vsaka tabela ima svoj indeks in ker smo uvozili eno samo ima ta prav gotovo začetni indeks 0.

```
//POVEZOVALNI NIZ Z ZBIRKO PODATKOV
string connectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NabavaSQL.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True";
private void Form1_Load(object sender, EventArgs e)
{
    try
    {
        SqlDataAdapter daDobavitelji;//adapter za prenos podatkov o dobaviteljih
        DataSet dsNabavaSQL; //DataSet za shranjevanje podatkov v pomnilniku
        string poizvedba = "SELECT * FROM Dobavitelji";
        //Objekt tipa SqlAdapter vzpostavi komunikacijo z bazo
        daDobavitelji = new SqlDataAdapter(poizvedba, connectionString);
        dsNabavaSQL = new DataSet();
        daDobavitelji.Fill(dsNabavaSQL, "Dobavitelji");
        //objekt comboBox1 povežemo s podatkovnim izvorom
        comboBox1.DataSource = dsNabavaSQL.Tables[0];
        //prikazana bodo imena dobaviteljev
        comboBox1.DisplayMember = "Ime";
        //ob izbiri dobavitelja bo "vrednost" izbire številka tega dobavitelja
        comboBox1.ValueMember = "ID_Dobavitelj";
        //klic metode, ki se sicer zgodi ob zapiranju gradnika ComboBox
        comboBox1_DropDownClosed(this, null);

        ////ALI PA Z UPORABO using stavka
        /*DataSet dsNabavaSQL; //DataSet za shranjevanje podatkov v pomnilniku
        string poizvedba = "SELECT * FROM Dobavitelji";
        using (SqlDataAdapter daDobavitelji = new SqlDataAdapter(poizvedba,
connectionString))
        {
            dsNabavaSQL = new DataSet();
            daDobavitelji.Fill(dsNabavaSQL, "Dobavitelji");
            comboBox1.DataSource = dsNabavaSQL.Tables[0];
            comboBox1.DisplayMember = "Ime";
            comboBox1.ValueMember = "ID_Dobavitelj";
            comboBox1_DropDownClosed(this, null);
        }*/
    }
    catch
    {
        MessageBox.Show("Napaka pri dostpu do baze!\nProjekt se bo zaprl!");
        Application.Exit();
    }
}
//metoda, ki se izvede, ko uporabnik izbere dobavitelja v gradniku ComboBox
private void comboBox1_DropDownClosed(object sender, EventArgs e)
{
    DataSet dsNabavaSQL; //DataSet za shranjevanje prenešenih podatkov v
pomnilniku
    SqlDataAdapter daArtikli;//adapter za prenos podatkov o artiklih
    string poizvedba="SELECT * FROM Artikli WHERE ID_Dobavitelj='" +
comboBox1.SelectedValue.ToString() + "'";
    daArtikli = new SqlDataAdapter(poizvedba, connectionString);
    dsNabavaSQL = new DataSet();
    daArtikli.Fill(dsNabavaSQL, "Artikli");
    //objekt dataGridView1 povežemo s podatkovnim izvorom
```

```
    dataGridView1.DataSource = dsNabavaSQL.Tables[0];
}
```

Povezava med dvema tabelama s pomočjo gradnika *DataSet*

Povezavo realizirajmo še s pomočjo gradnika *DataSet*. V projekt dodajmo gradnik *DataSet* (to smo v tem poglavju že počeli), ga poimenujmo *NabavaSQLDataSet* in kliknemo gumb *Add*. Pojavi se prazen *DataSet* v katerem izberemo *DataBase Explorer*, da se na levi strani prikaže okno *DataBase Explorer* in v njem vrstica *DataConnection*. Desno kliknimo na *DataConnection* in izberemo *Add Connection*. Odpre se okno *Add Connection*, v katerem za *Datasource* izberemo *Microsoft SQL Server Database file*, s klikom na gumb *Browse* pa nato piščemo našo bazo *NabavaSQL*. Izbiro potrdimo s klikom na gumb *Open*. V oknu *Database Explorer* se pojavi vrstica *NabavaSQL.mdf*. Razširimo jo, nato razširimo še vrstico *Tables* in v okno *NabavaDataSet* povlecimo tabeli *Artikli* in *Dobavitelji*. Obakrat se na ekranu pojavi sporočilno okno, v katerem izberemo opcijo *Da* – bazo bomo za potrebe razvoja aplikacije prekopirali v naš projekt. Preklopimo na vizuelni pogled obrazca in gradniku *ComboBox* nastavimo tri lastnosti:

- lastnost *DataSource* nastavimo na *Other Data Sources* → *Project Data Sources* → *NabavaDataSet* → *Dobavitelji*;
- lastnost *DisplayMember* nastavimo na *Ime*;
- lastnost *ValueMember* nastavimo na *ID_Dobavitelj*.

Povezava prvega gradnika je gotova. Če projekt poženemo, so v *ComboBox*-u imena vseh dobaviteljev iz tabele *Dobavitelji* baze *NabavaSQL*.

Izbiro dobavitelja v gradniku *ComboBox* moramo sedaj še vezati na prikaz artiklov v gradniku *DataGridView*. Gradnik *DataGridView*, ki je že na obrazcu, preko lastnosti *DataSource* povežemo s tabelo *Artikli* (*Other Data Sources* → *Project Data Sources* → *NabavaDataSet* → *Artikli*). Ustvarimo še odzivno metodo dogodka *Shown* našega obrazca. Vanjo zapišimo *SELECT* stavek, ki bo poskrbel za prenos artiklov le tistega dobavitelja, ki bo izbran v gradniku tipa *ComboBox*. .

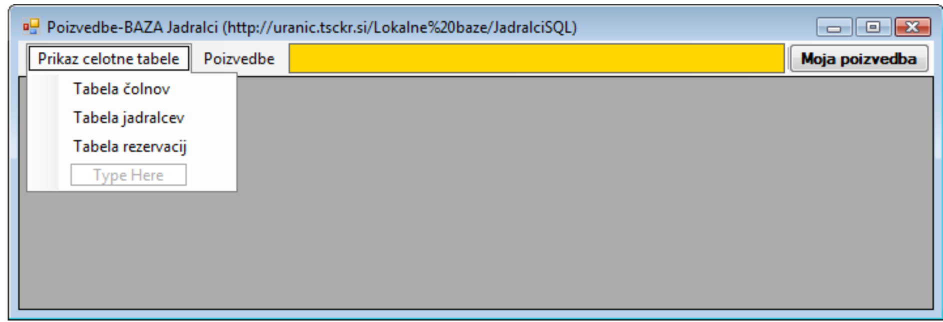
```
private void Form1_Shown(object sender, EventArgs e)
{
    /*s spremembo SELECT stavka dosežemo, da so v gradniku DataGridView
      prikazani le artikli dobavitelja, ki je izbran v gradniku ComboBox.  */
    artikliTableAdapter.Adapter.SelectCommand.CommandText = "SELECT * FROM
Artikli WHERE ID_Dobavitelj='" + comboBox1.SelectedValue.ToString() + "'";
    this.artikliTableAdapter.Fill(this.nabavaSQLDataSet.Artikli);
}
```

Ostane le še odzivna metoda dogodka *DropDownClosed* gradnika *ComboBox*. Izberimo *ComboBox* in v oknu *Properties* odprimo spustni seznam pri tem dogodku. Izberemo že ustvarjeno odzivno metodo *Form1*_Shown. Po vsakem zapiranju gradnika *ComboBox* se izvede nov *SELECT* stavek, ki ima za posledico prikaz artiklov izbranega dobavitelja v gradniku *DataGridVew*.

Poizvedbe

Na strežniku http://uranic.tsckr.si/Lokalne%20baze/JadralciSQL je že pripravljena baza *JadralciSQL* v kateri so tri tabele: *Coln*, *Jadralec* in *Rezervacija*. Iz te baze bi radi naredili nekaj različnih poizvedb in rezultate prikazali v gradniku *DataGridView*.

V projekt najprej vključimo *using* stavek *System.Data.SqlClient*. Na obrazec postavimo gradnik *MenuStrip*, v katerem so tri menijske postavke: dve postavki sta tipa *MenuItem* (*Prikaz celotne tabele* in *Poizvedbe*), tretja pa je tipa *TextBox*, kamor bomo lahko zapisali poljubno poizvedbo. To poizvedbo bomo ustvarili s pomočjo gumba z napisom *Moja poizvedba*, ki ga postavimo ob *TextBox*. Preostali del obrazca zapolnjuje gradnik tipa *DataGridView*.



Slika 141: Obrazec za izdelavo poizvedb iz baze *JadralciSQL*.

Postavki glavnega menija naj vsebujeta naslednje poizvedbe:

- Prikaz celotne tabele
  - Tabela čolnov.
  - Tabela jadralcev.
  - Tabela rezervacij.

- Poizvedbe
  - Jadralci starejši od 30 let.
  - Vsi podatki o rdečih čolnih.
  - Seznam rezervacij jadralcev starejših od 50 let in njihovih čolnov.
  - ID čolnov, ki jih je rezerviral Darko.
  - Dolžina čolna z imenom Bavaria

Za prvo poizvedbo (*Tabela vseh čolnov*) napišimo odzivno metodo, v katero zapišimo vse potrebne stavke za realizacijo te poizvedbe.

```
//Povezovalni niz z bazo podatkov
string connectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\DOKUMENTI\APJ+ŠOLA\APJ\SQL
SERVER\JadralciSQL\Jadralci.mdf;Integrated Security=True;User Instance=True";
```

```
//Odzivna metoda menijske vrstice za prikaz tabele vseh čolnov
private void tabelaToolStripMenuItem_Click(object sender, EventArgs e)
{
    try
    {
        //besedilo poizvedbe
        string poizvedba = "select * from Coln";
        //vzpostavimo komunikacijo z bazo in izvedemo poizvedbo
        SqlDataAdapter DAJadralci = new SqlDataAdapter(poizvedba,
connectionString);

        //Podatke iz tabele Coln uvozimo v objekt tipa DataTable
        DataTable table = new DataTable();

        /*Nastavitev lokalnih informacij za primerjavo nizov znotraj
          objekta table*/
        table.Locale = System.Globalization.CultureInfo.InvariantCulture;

        //objekt table napolnimo s podatki iz baze
        DAJadralci.Fill(table);
            // ali  tudi takole
            ////DataSet dsJadralci=new DataSet();
            ////DAJadralci.Fill(dsJadralci,"Coln");
            ////dataGridView1.DataSource = dsJadralci.Tables[0];

        /*Dimenzijo stolpcev v DataGridView prilagodimo glede na vsebino*/
dataGridView1.AutoResizeColumns(DataGridViewAutoSizeColumnsMode.AllCellsExcep
tHeader);

        //Podatki v gradniku DataGridView naj bodo ReadOnly
        dataGridView1.ReadOnly = true;
        //za boljši pregled naj bo ozadje sodih vrstic obarvano drugače
        dataGridView1.AlternatingRowsDefaultCellStyle.BackColor =
              Color.LightSteelBlue;
        //Gradnik DataGridView še povežemo s podatkovnim izvorom
        dataGridView1.DataSource = table;

        //Poljuben stolpec lahko skrijemo: skrijemo npr. prvi stolpec
        dataGridView1.Columns[0].Visible = false;
        //Širina stolpcev naj bo npr. taka, da zapolnijo celotno tabelo
        for (int i = 0; i < dataGridView1.Columns.Count; i++)
            dataGridView1.Columns[i].Width = this.Width
               /(dataGridView1.Columns.Count-1);
    }
    catch
    {
        MessageBox.Show("Napaka pri dostopu do baze podatkov!");
    }
}
```

Napisali smo tudi nekaj stavkov, v katerih smo poskrbeli za lepši prikaz podatkov v celicah gradnika *DataGridView*. Dimenzije stolpcev smo prilagodili glede na vsebino, sode vrstice pa so

zaradi boljšega pregleda drugače pobarvane. Širine stolpcev smo določili tako, da bodi vsi stolpci skupaj ravno zapolnili celotno širino tabele.

Za ostale poizvedbe napišimo svojo metodo *Poizvedba*, ki ji bomo za parameter posredovali niz poizvedbe, metoda pa bo rezultat poizvedbe prikazala v gradniku *DataGridView*. V metodo vključimo varovalni blok. Ta bo v primeru napačne poizvedbe, ali pa karšnekoli napake pri komunikaciji z bazo, izvrgel obvestilo o napaki.

```
//Metoda za izdelavo poizvedb - vhodni podatek je ustrezna poizvedba
private void Poizvedba(string SQLPoizvedba)
{
    try
    {
        dataGridView1.DataSource = "";
        //nizu za povezavo z bazo priredimo vrednost parametra metode
        string poizvedba  = SQLPoizvedba;

        //vzpostavimo komunikacijo z bazo in izvedemo poizvedbo
        SqlDataAdapter dataAdapter = new SqlDataAdapter(poizvedba,
connectionString);
        DataTable table = new DataTable();
        dataAdapter.Fill(table);

        //za boljši pregled naj bo ozadje sodih vrstic obarvano drugače
        dataGridView1.AlternatingRowsDefaultCellStyle.BackColor =
Color.LightSteelBlue;

        //Podatki v gradniku DataGridView naj bodo ReadOnly
        dataGridView1.ReadOnly = true;
        //povezava gradnika s podatkovnim izvorom
        dataGridView1.DataSource = table;

        //Širina stolpcev naj bo taka, da zapolnijo celoten DataGridView
        for (int i = 0; i < dataGridView1.Columns.Count; i++)
            dataGridView1.Columns[i].Width = this.Width
/dataGridView1.Columns.Count;
    }
    catch
    {
        MessageBox.Show("Napaka pri dostopu do baze podatkov!");
    }
}
```

Večino ostalih poizvedb v meniju naredimo tako, da besedilo poizvedbe posredujemo metodi *Poizvedbe*.

```
//Poizvedba za seznam vseh jadralcev iz tabele Jadralec
private void tabelaToolStripMenuItem1_Click(object sender, EventArgs e)
{
    Poizvedba("select * from Jadralec");
}
//Poizvedba za rezervacijami iz tabele Rezervacija
```

```
private void tabelaRezervacijToolStripMenuItem_Click(object sender, EventArgs
e)
{
    Poizvedba("select * from Rezervacija");
}
//Poizvedba za jadralci, ki so starejši od 30 let
private void jadralciStarejšiOd30LetToolStripMenuItem_Click(object sender,
EventArgs e)
{
    Poizvedba("select * from Jadralec where starost>30");
}
//Poizvedba o vseh rdečih čolnih
private void vsiPodatkiORdečihČolnihToolStripMenuItem_Click(object sender,
EventArgs e)
{
    Poizvedba("select *from coln where barva='rdeca'");
}
//Poizvedba za rezervacije jadralcev starejših od 50 let in njihovih čolnov
private void
imenaJadralcevInČolnovRezervacijeJadralcevStarejšihOd50LetToolStripMenuItem_C
lick(object sender, EventArgs e)
{
    Poizvedba("select jadralec.ime,Coln.ime from jadralec,coln,rezervacija
where jadralec.jid=rezervacija.jid and rezervacija.cid=coln.cid  and
Jadralec.starost>50");
}

//Poizvedba za vsemi čolni, ki jih je rezerviral Darko
private void iDČolnovKiJihJeRezervToolStripMenuItem_Click(object sender,
EventArgs e)
{
    Poizvedba("Select rezervacija.cid,jadralec.ime from rezervacija JOIN
jadralec ON jadralec.jid=rezervacija.jid where ime='Darko'");
}
```

Nekoliko drugačna je poizvedba o dolžini čolna z imanom 'Bavaria'.

```
try
{
    //povezem se na bazo
    dataConnection = new SqlConnection(connectionString);
    SqlCommand ukaz = new SqlCommand();
    ukaz.Connection = dataConnection;
    ukaz.CommandType = CommandType.Text;
    string ime = "Bavaria";//ime čolna, katerega dolžino bomo poiskali
    //besedilo poizvedbe
    ukaz.CommandText = "SELECT dolzina FROM coln where Ime = '" + ime + "';";
    dataConnection.Open();
    Object rezultat = ukaz.ExecuteScalar();//rezultat shranimo v objekt
    dataConnection.Close();
    MessageBox.Show("Dožina čolna Bavaria: "+ (int)rezultat);
}
catch (Exception ex)
```

```
{
    MessageBox.Show(ex.Message);
}
```

Ostane nam še odzivna metoda gumba z napisom *Poizvedba*. Uporabnik namreč v tekstovno polje levo od tega gumba lahko zapiše poljubno svojo poizvedbo, ki se bo izvedla ob kliku na ta gumb. Za poizvedbo zopet uporabimo klic metode *Poizvedba*.

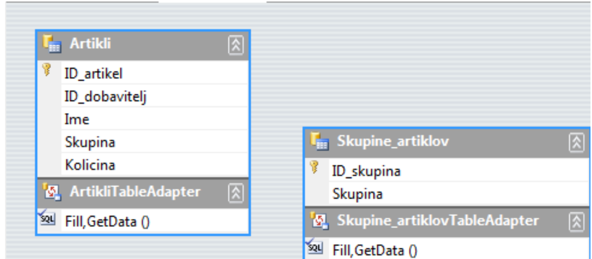
```
//Klic poizvedbe, ki jo vpiše uporabnik v TextBox
private void bPoizvedba_Click(object sender, EventArgs e)
{
    Poizvedba(toolStripTextBox1.Text);
}
```

Transakcije

Kadar je operacija nad podatki še posebno pomembna ali pa sestavljena iz več ukazov, uporabimo transakcijo. To se zgodi pri dodajanju novih zapisov, pri ažuriranju podatkov ali pa pri brisanju podatkov. S transakcijami tudi rešimo težave, ki bi jij sicer lahko imeli v primeru, ko ima brisanje zapisa v eni tabeli za posledico hkratno brisanje določenih zapisov v drugi tabeli. Transakcija običajno (ni pa nujno) povzroči več sprememb v eni ali pa v več tabelah. Predstavlja tudi najboljši način za dodajanje novega zapisa v tabelo ali pa za ažuriranje zapisa v tabeli. Bistvena lastnost transakcij je, da se bodo izvedli vsi *SQL* ukazi znotraj transakcije, ali pa nobeden. Če se torej transakcija ne izvede v celoti, se vse spremembe zavržejo, vzpostavi se prvotno stanje, to je stanje pred transakcijo. Uporabljamo jih torej za zavarovanje podatkov v vsakem primeru, ko obstaja možnost, da med izvajanjem *SQL* stavkov pride do napake in je potrebno vzpostaviti prvotno stanje.

Transakcijo pričnemo z ustvarjanjem novega objekta razreda *SQLTransaction*, končamo pa s stavkom *COMMIT*, ki dokončno potrdi spremembe v bazi, ali pa z ukazom *ROLLBACK* (ki spremembe zavrže).

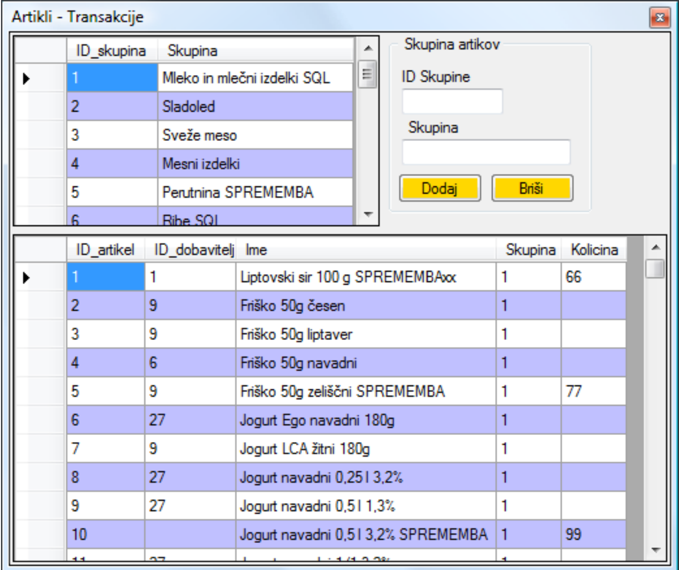
Uporabo transakcij bomo prikazali v naslednjem projektu. V nov projekt dodajmo *DataSet* in ga povežimo z bazo *nabavaSQL*. V *DataSet* dodajmo dva *TableAdapter*-ja za tabeli *Artikli* in *Skupina_artiklov*.



Slika 142: *Dataset* z dvema tabelama.

Na osnovni obrazec dodajmo dva gradnika *DataGridView* in jima določimo lastnost *DataSource* tako, da bomo imeli v zgornjem gradniku (objekt *dataGridView1*) vsebino tabele *Skupine_artiklov*, v spodnjem gradniku (objekt *dataGridView2*) pa vsebino tabele *Artikli*. Na

obrazec postavimo še gradnik *GroupBox* in vanj postavimo dva gradnika *Label*, dva gradnika *TextBox* in dva gumba (*bDodaj* in *bBrisi*).



Slika 143: Obrazec za prikaz transakcij.

Odzivni metodi dogodka *Click* gumba *bDodaj* bomo priredili transakcijo, ki bo v tabelo *Skupina_artiklov* dodala nov zapis (potrebna podatka vnesemo v oba gradnika *TextBox*).

V *Insert* stavek znotraj transakcije bomo vrednosti novih podatkov, ki jih želimo dodati v tabelo, prenesli s pomočjo *Sql* paramatrov. Nov *Sql* parameter določimo takole:

```
SqlCommand dataCommand = new SqlCommand();//Objekt tipa SqlCommand
dataCommand.Parameters.AddWithValue("@imeParametra", imeObjekta);
```

Sql parameter določimo im mu dodelimo vrednost s pomočjo metode *AddWithValue*, ki pripada objektu *dataCommand*, izpeljanemu iz razreda *SqlCommand*. Ime *Sql* parametra je poljubno, oznaka "@" pred imenom pa označuje, da gre za *Sql* parameter. Drugi parameter metode je vrednost, ki jo želimo prirediti *Sql* parametru. Za vstavljanje, ali pa ažuriranje podatkov nekega zapisa potrebujemo metodo *ExecuteNonQuery*. Z metodo *Commit* skušamo dokončno potrditi spremembe va tabeli.

```
private void bDodaj_Click(object sender, EventArgs e)
{
    SqlConnection dataConnection = new SqlConnection();
    dataConnection.ConnectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\DOKUMENTI\APJ+ŠOLA\APJ\SQL
SERVER\NabavaSQL\NabavaSQL.mdf;Integrated Security=True;Connect
Timeout=30;User Instance=True";
    dataConnection.Open();
    //Ustvarimo nov objekt za transakcijo
    SqlTransaction transakcija = dataConnection.BeginTransaction();
    //Ustvarimo nov objekt za SQL ukaz
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = dataConnection;
```

```
    dataCommand.Transaction = transakcija;
    try
    {
        int novID = Convert.ToInt32(textBox1.Text);
        string novaSkupina = textBox2.Text;
        //nove vrednosti dodamo v tabelo preko Sql parametrov @ID in @novaSK
        dataCommand.Parameters.AddWithValue("@ID", novID);
        dataCommand.Parameters.AddWithValue("@novaSk", novaSkupina);
        dataCommand.CommandText = "Insert INTO Skupine_artiklov
(ID_skupina,Skupina) VALUES (@ID,@novaSk)";

        /*ali pa tudi takole
        dataCommand.Parameters.Add(newSqlParameter("@ID",textBox1.Text));
        dataCommand.Parameters.Add(new SqlParameter("@novaSK",
                                     textBox2.Text));
        dataCommand.CommandText = "Insert INTO Skupine_artiklov
               (ID_skupina,Skupina) VALUES (@ID,@novaSk)";*/

        /*lahko pa tudi brez uporabe parametrov takole:
        dataCommand.CommandText = "Insert INTO Skupine_artiklov
    (ID_skupina,Skupina) VALUES ('"+textBox1.Text+"','"+textBox2.Text+"');";*/

        dataCommand.CommandType = CommandType.Text;
        dataCommand.ExecuteNonQuery();
        //dokončno potrdimo spremembe v bazi
        transakcija.Commit();
        MessageBox.Show("Dodajanje nove skupine uspešno!");
    }
    catch (Exception ep)
    {
        //Če je prišlo do napake, v bazi vzpostavimo prvotno stanje
        transakcija.Rollback();
        MessageBox.Show("Napaka pri shranjevanju podatkov!");
    }
    finally
    {
        dataConnection.Close();
        //Ažuriramo vsebino dataSet1 in s tem tudi gradnika dataGridView1
this.skupine_artiklovTableAdapter.Fill(this.dSNabavaSQL.Skupine_artiklov);
        this.artikliTableAdapter.Fill(this.dSNabavaSQL.Artikli);
    }
}
```

Odzivni metodi dogodka *Click* gumba *Briši* pa bomo priredili transakcijo, ki bo v tabeli *Artikli* pobrisala vse zapise, v katerih nastopa podatek *ID Skupine*, ki ga zapišemo v gradnik *textBox1*.

```
private void bBrisi_Click(object sender, EventArgs e)
{
    SqlConnection dataConnection = new SqlConnection();
    dataConnection.ConnectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\DOKUMENTI\APJ+ŠOLA\APJ\SQL
SERVER\NabavaSQL\NabavaSQL.mdf;Integrated Security=True;Connect
Timeout=30;User Instance=True";
```

```
    dataConnection.Open();
    //Ustvarimo nov objekt za SQL transakcijo v SQL bazi podatkov
    SqlTransaction myTrans = dataConnection.BeginTransaction();
    //Ustvarimo nov objekt za SQL stavek, ki ga bomo izvedli v SQL bazi
    SqlCommand dataCommand = new SqlCommand();
    dataCommand.Connection = dataConnection;
    dataCommand.Transaction = myTrans;
    try
    {
        //shranimo vrednost v celici ID_skupina izbrane vrstice
        int ID = Convert.ToInt32(textBox1.Text);
        /*@ID je parameter, preko katerega bomo transakciji posredovali
         vrednost spremenljike ID*/
        dataCommand.Parameters.AddWithValue("@ID", ID);
        /*oblikujemo SQL ukaz za brisanje vseh artiklov iz tabele Artikli,
          ki pripadajo skupini zapisani v textBox1*/
        dataCommand.CommandText = "delete Artikli where Skupina=@ID";
        //Poizkusimo izvesti dejansko brisanje vseh artiklov iz te skupine
        dataCommand.ExecuteNonQuery();
        myTrans.Commit();//dokončno shranimo narejene spremembe
    }
    catch (Exception ep)
    {
        myTrans.Rollback();//v primeru napake se vzpostavi prejšnje stanje
        MessageBox.Show("Napaka pri brisanju podatkov!");
    }
    finally
    {
        dataConnection.Close();//na koncu zapremo povezavo z bazo podatkov
        this.artikliTableAdapter.Fill(this.dSNabavaSQL.Artikli);
    }
}
```

Kadar bomo pisali aplikacijo namenjeno uporabnikom, ki bodo do nje dostopali npr. preko omrežnih povezav ali celo preko interneta (*remote users*), bomo v stavku *ConnectionString* zapisali tudi uporabniško ime in geslo, npr. takole:

```
string connetionString = "Data Source=ServerName;Initial
Catalog=DatabaseName;User ID=UserName;Password=Password";
```

Povzetek

Projekti, ki delajo z bazo podatkov so med najzahtevnejšimi. Pomagamo si lahko s čarovnikom, ki preko gradnika *DataSet* in številnih vgrajenih razredov omogoča dokaj lagodno manipulacijo s tabelami poljubne baze podatkov. Spoznali pa smo tudi transakcije, ki predstavljajo najbolj

zanesljiv način za urejanje tabel in za dodajanje novih zapisov. Pri ustvarjanju novih projektov nam bodo opisani postopki in metode v veliko pomoč, glede na naravo problema, ki ga moramo rešiti, pa se bomo odločili med uporabo čarovnika in transakcijskim modelom.

Države

Pridobljeno znanje za delo z bazo podatkov bomo sedaj uporabili za izdelavo večokenskega projekta *DržaveNaSvetu*, v katerem bomo prikazali popolno manipulacijo z bazo *DrzaveSQL*. Projekt bomo izdelali povsem programsko, torej brez uporabe gradnika *DataSet* oz. brez uporabe čarovnikov. Po navodilih iz začetka poglavja o bazah najprej ustvarimo bazo *DrzaveSQL* z dvema tabelama (*Drzave* in *Kontinenti*).

Struktura tabele *Drzave* naj bo takale:

- ID_Drzava: int (ključno polje*, Identity Specification* nastavimo na *Yes*, kar pomeni, da je polje *Autoincrement*, oz. da se bo *ID*- številka države določala avtomatično in sicer od 1 naprej, (*Allow Nulls:* NE);
- Drzava: nvarchar(255), (*Allow Nulls:* DA);
- Kontinnet: nvarchar(255), (*Allow Nulls:* DA);
- Povrsina: float, (*Allow Nulls:* DA);
- Prebivalstvo: float, (*Allow Nulls:* DA);
- BDP: float, (*Allow Nulls:* DA).

Struktura tabele *Kontinent*:

- ID_Kontinenta: int (ključno polje*, Identity Specification* naj bo *Yes, Allow Nulls*: NE);
- Ime: nvarchar(255) (*Allow Nulls*: NE).

V obe tabeli že med njunim ustvarjanjem vnesimo nekaj testnih podatkov, npr:

SRECO-PC\...\DRZ... - dbo.Kontinenti

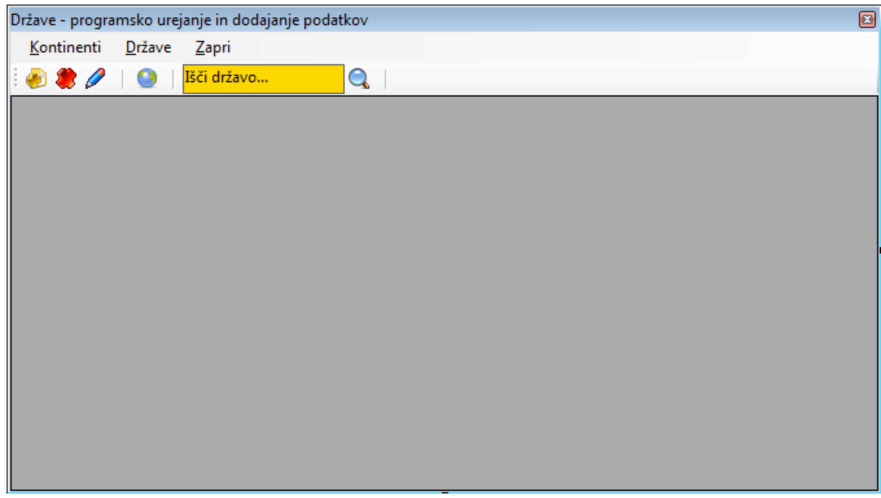
ID_Kontinenta	Ime
1	Evropa
2	Afrika
3	Avstralija
4	Oceanija
5	Azija
6	Severna Amerika
7	Južna Amerika
8	Srednja Amerika
9	Antarktika

SRECO-PC\...\DRZ....MDF - dbo.Drzave | SRECO-PC\...\DRZ... - dbo.Kontinenti | SRECO-PC\...\DRZ....MDF - dbo.Drzave

ID_Drzava	Drzava	Kontinent	Povrsina	Prebivalstvo	BDP
2	Združeno kraljes...	Evropa	244820	59511464	1290000000000
3	Združeni Arabski...	Azija	82880	2369153	41500000000
4	Združene držav...	Severna Amerika	9629091	275562673	9255000000000
5	Zambija	Afrika	752614	9582418	8500000000
6	West Bank	Azija	5860	2020298	3300000000
7	Vietnam	Azija	329560	78773873	143100000000
8	Venezuela		912050	23542649	182800000000

Slika 144: Testni podatki v tabelah *Kontinenti* in *Drzave* baze *DrzaveSQL*.

Glavni obrazec projekta se imenuje *FDrzave*, vsebuje naj menijsko vrstico in orodjarno, pod njima pa postavimo gradnik *DataGridView*, ki mu lastnost *Dock* nastavimo na *Fill*.



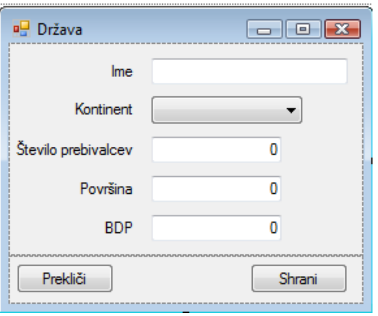
Slika 145: Glavni obrazec projekta *DržaveNaSvetu.*

Vrstica z menijem vsebuje tri postavke: *Kontinenti*, *Države*, ter *Zapri*. Postavka *Države* naj vsebuje še podmeni z možnostmi *Dodaj državo*, *Uredi izbrano državo* in *Briši izbrano državo*. Gumbi v orodjarni so le bližnjice za postavke v menijski vrstici (*Dodaj državo*, *Brisanje izbrane države*, *Urejanje izbrane države* in *Kontinenti*). Dodano je še vnosno polje za vpis imena države in gumb za iskanje te države v tabeli.

Projekt vsebuje še tri podobrazce. Ker pa želimo v podobrazcih dostopati do objektov glavnega obrazca, moramo najprej ustrezno spremeniti projektno datoteko *Program.cs*.

```
class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    // NAPOVED novega objekta tipa FDrzave (objekt bo STATIČEN)
    public static FDrzave drzave;
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        //ustvarjanje novega objekta tipa Fdrzave
        drzave = new FDrzave();
        Application.Run(drzave);
    }
}
```

Pred pisanjem odzivnih metod, ustvarimo še podobrazec za dodajanje oz. ažuriranje nove države (*Drzava.cs).* Vsi gradniki naj imajo lastnost *Modifiers* nastavljeno na *Public*. V gradnik tipa *ComboBox* bomo s pomočjo poizvedbe uvozimli podatke iz tabele *Kontinenti*. Prikazano je polje *ime* kontinenta, lastnost *ValueMember* gradnika pa je nastavljena na polje *Id_Kontinenta*. Gumba na obrazcu sta modalna: gumb *Shrani* vrne vrednost *DialogResult.OK*, gumb Pekliči pa *DialogResult.Cancel*.



Slika 146: Podobrazec za dodajanje nove države in za ažuririranje podatkov o izbrani državi.

Lotimo se sedaj odzivnih metod glavnega obrazca. V njem najprej napovemo vse potrebne objekte za delo z bazo podatkov. Zapišimo tudi povezovalni niz. Uporabljali ga bomo v vseh obrazcih našega projekta, zato ga ustvarimo izven vseh metod. Povezavo z bazo in prikaz celotne tabele *Drzave* bomo na glavnem obrazcu ustvarili s pomočjo konstruktorja. Dodana je še odzivna metoda za zapiranje obrazca.

```
Public partial class Fdrzave : Form
{
    SqlConnection dataConnection;
    SqlDataAdapter daDrzave;
    /*objekti za povezavo z bazo so javni zato, da jih lahko uporabimo v
     podrejenih obrazcih*/
    public DataSet dsDrzave;
    //povezovalni niz
    public string izvor = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\DOKUMENTI\APJ+ŠOLA\APJ\SQL
SERVER\DrzaveSQL\DrzaveSQL.mdf;Integrated Security=True;Connect
Timeout=30;User Instance=True";
    //Konstruktor obrazca
    public Fdrzave()
    {
        InitializeComponent();
        try
        {
            //Ko se obrazec odpre bo v tabeli že seznam vseh vnesenih držav
            string SQL = "select * from Drzave";
            dataConnection = new SqlConnection(izvor);
            dsDrzave = new DataSet();
            daDrzave = new SqlDataAdapter(SQL, dataConnection);
            daDrzave.Fill(dsDrzave, "Drzava");
            dataGridView1.DataSource = dsDrzave.Tables[0];
        }
        catch
        {
            MessageBox.Show("Napaka pri dostopu do baze podatkov! ");
        }
    }
    //ZAPIRANJE obrazca
    private void zapriToolStripMenuItem_Click(object sender, EventArgs e)
```

```
    {
        Close();
    }
}
```

V odzivna metoda za dodajanje nove države najprej ustvarimo nov objekt razreda *Drzava*. Tako ustvarjen obrazec odpremo modalno. če je uporabnik vanj uspešno vnesel podatke, ustvarimo transakcijo tipa *insert*. Za dodajanje novih podatkov v tabelo potrebujemo kar nekaj *Sql* parametrov, ki jim bomo dali imena, ki nas bodo asociirala na njihov pomen. Z metodo *ExecuteNonQuery* sprožimo vstavljanje v tabelo, z metodo *Commit* pa dokončno potrditi narejene spremembe va tabeli.

```
//DODAJANJE nove države
private void dodajToolStripMenuItem_Click(object sender, EventArgs e)
{
    Drzava Nova = new Drzava();
    Nova.Text = "Vnos nove države!";
    //Ko se obrazec odpre, bo v spustnem seznamu že prvi kontinent
    Nova.comboBox1.SelectedIndex = 0;
    if (Nova.ShowDialog() == DialogResult.OK)
    {
        //Pred začetkom transakcije odpremo povezavo z bazo podatkov
        dataConnection.Open();
        SqlTransaction myTrans = dataConnection.BeginTransaction();
        SqlCommand dataCommand = new SqlCommand();
        dataCommand.Connection = dataConnection;
        dataCommand.Transaction = myTrans;
        try
        {
            //deklaracija SQL parametrov za SQL transacijo
            dataCommand.Parameters.Add(new SqlParameter("@Drzava",
                 Nova.textBox1.Text));
            dataCommand.Parameters.Add(new SqlParameter("@Kontinent",
                 Nova.comboBox1.Text));
            dataCommand.Parameters.Add(new SqlParameter("@Povrsina",
                 Nova.textBox2.Text));
            dataCommand.Parameters.Add(new SqlParameter("@Prebivalstvo",
                 Nova.textBox3.Text));
            dataCommand.Parameters.Add(new SqlParameter("@BDP",
                 Nova.textBox4.Text));
            //ukazni niz za vstavljanje novega zapisa v tabelo
            dataCommand.CommandText = "Insert INTO Drzave (Drzava,Kontinent,
Povrsina,Prebivalstvo,BDP) VALUES (@Drzava,@Kontinent,@Povrsina,@Prebivalstvo
,@BDP)";
            dataCommand.CommandType = CommandType.Text;
            //za zapis v tabelo potrebujemo metodo ExecuteNonQuery)
            dataCommand.ExecuteNonQuery();
            myTrans.Commit();//poizkus zapisa v tabelo
        }
        catch
        {
            //Če je prišlo do napake, vzpostavimo prejšnje stanje
            myTrans.Rollback();
```

```
            MessageBox.Show("Šifra skupine že obstaja ali pa napaka pri 
shranjevanju!");
        }
        string SQL = "SELECT * FROM Drzave";
        dataConnection = new SqlConnection(izvor);
        daDrzave = new SqlDataAdapter(SQL, dataConnection);
        dsDrzave = new DataSet();
        daDrzave.Fill(dsDrzave, "Drzava");
        dataGridView1.DataSource = dsDrzave.Tables[0];
    }
}
```

Po uspešno izvedeni transakciji smo poskrbeli za osveževanje vsebine gradnika *DataGridView*, v katerem se pojavi novo vnesena država.

Za ažuriranje podatkov izbranega zapisa prav tako uporabimo objekt razreda *Drzava*. Razlika med dodajanjem in ažuriranjem pa je seveda v tem, da moramo pred prikazom obrazca za ažuriranje poskrbeti, da so na njem podatki izbranega zapisa. Ker smo vsem vizuelnim objektom na obrazcu *Drzava* nastavili lastnost *Modifiers* na *Public*, to sedaj ne bo težko. Iz osnov *OOP* in poglavja o večokenskih aplikacijah vemo, da do njih dostopamo s pomočjo operatorja pika. Vrednosti podatkov bomo pobirali iz celic in jih kasneje prav tako shranjevali v celice gradnika *DataGridView*. To pa smo se naučili že v poglavju, v katerem smo podrobno predstavili ta gradnik.

Tokratna transakcija bo tipa *update*. Po uspešni izvedbi moramo še poskrbeti za osveževanje vsebine gradnika *DataGridView*. Podatki države, ki smo jih ravnokar spreminjali, bodo spremenejni tudi na ekranu.

```
//AŽURIRANJE podatkov o izbrani državi pri dvokliku na DataGridView
private void dataGridView1_DoubleClick(object sender, EventArgs e)
{
    Drzava Azuriraj = new Drzava();
    Azuriraj.Text = "Ažuriranje podatkov izbrane države države!";
    //podatek izbrane vrstice gradnika DataGridView prenesemo na obrazec
    Azuriraj.textBox1.Text = Convert.ToString(
                               dataGridView1.CurrentRow.Cells[1].Value);
    int poz=Azuriraj.comboBox1.FindString(Convert.ToString(
            dataGridView1.CurrentRow.Cells[2].Value),0);
    Azuriraj.comboBox1.SelectedIndex = poz;
    Azuriraj.textBox2.Text = Convert.ToString(
                               dataGridView1.CurrentRow.Cells[3].Value);
    Azuriraj.textBox3.Text = Convert.ToString(
                               dataGridView1.CurrentRow.Cells[4].Value);
    Azuriraj.textBox4.Text = Convert.ToString(
                               dataGridView1.CurrentRow.Cells[5].Value);
    int pozicija = Convert.ToInt32(dataGridView1.CurrentRow.Cells[0].Value);
    if (Azuriraj.ShowDialog() == DialogResult.OK)
    {
        //Pred začetkom transakcije odpremo povezavo z bazo podatkov
        dataConnection.Open();
        SqlTransaction myTrans = dataConnection.BeginTransaction();
```

```
        SqlCommand dataCommand = new SqlCommand();
        dataCommand.Connection = dataConnection;
        dataCommand.Transaction = myTrans;
        try
        {
            //deklaracija parametrov za SQL transacijo
            dataCommand.Parameters.Add(new SqlParameter("@ID_Drzava",
               pozicija));
            //ali lahko tudi:
            dataCommand.Parameters.AddWithValue("@ID_Drzava",pozicija);
            dataCommand.Parameters.Add(new SqlParameter("@Drzava",
               Azuriraj.textBox1.Text));
            dataCommand.Parameters.Add(new SqlParameter("@Kontinent",
               Azuriraj.comboBox1.Text));
            dataCommand.Parameters.Add(new SqlParameter("@Povrsina",
               Azuriraj.textBox2.Text));
            dataCommand.Parameters.Add(new SqlParameter("@Prebivalstvo",
               Azuriraj.textBox3.Text));
            dataCommand.Parameters.Add(new SqlParameter("@BDP",
               Azuriraj.textBox4.Text));
            dataCommand.CommandText = "Update Drzave  SET Drzava=@Drzava,
                     Kontinent=@Kontinent,Povrsina=@Povrsina,Prebivalstvo=
                     @Prebivalstvo,BDP=@BDP WHERE ID_Drzava=@ID_Drzava";
            dataCommand.CommandType = CommandType.Text;
            dataCommand.ExecuteNonQuery();
            myTrans.Commit();//če vse OK, podatke dokončno shranimo v tabelo
        }
        catch
        {
            //Če je prišlo do napake, vzpostavimo prejšnje stanje
            myTrans.Rollback();
            MessageBox.Show("Napaka pri shranjevanju!");
        }
        string SQL = "SELECT * FROM Drzave";
        dataConnection = new SqlConnection(izvor);
        daDrzave = new SqlDataAdapter(SQL, dataConnection);
        dsDrzave = new DataSet();
        daDrzave.Fill(dsDrzave, "Drzava");
        dataGridView1.DataSource = dsDrzave.Tables[0];
    }
}
```

Dodajmo še odzivno metodo za brisanje zapisa iz tabele. Pred brisanjem uporabnika v sporočilnem oknu obvestimo, katero vrstica bo pobrisana. Če uporabnik brisanje potrdi naredimo transakcijo tipa *delete*. Po brisanju zopet poskrbimo za osveževanje zapisov na obrazcu.

```
//BRISANJE izbrane države
private void brisiToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (MessageBox.Show("Brišem državo " +
dataGridView1.CurrentRow.Cells[1].Value.ToString(), "Brisanje vrstice",
MessageBoxButtons.OKCancel, MessageBoxIcon.Question) == DialogResult.OK)
```

```
    {
        //zapomnim si šifro države, ki jo želim pobrisati
        int sifra = Convert.ToInt32(dataGridView1.CurrentRow.Cells[0].Value);
        dataConnection.Open();
        SqlTransaction myTrans = dataConnection.BeginTransaction();
        SqlCommand dataCommand = new SqlCommand();
        dataCommand.Connection = dataConnection;
        dataCommand.Transaction = myTrans;
        try
        {
            dataCommand.Parameters.Add(new SqlParameter("@ID_Drzava",sifra));
            //ali tudi takole:
            //SqlParameter param1 = new SqlParameter();
            //param1.ParameterName = "@ID_Drzava";
            //param1.Value = sifra;
            //dataCommand.Parameters.Add(param1);

            dataCommand.CommandText = "DELETE FROM Drzave WHERE ID_Drzava
                                       =@ID_Drzava";
            dataCommand.ExecuteNonQuery();
            myTrans.Commit();//če vse OK, podatke dokončno shranimo v tabelo
        }
        catch
        {
            myTrans.Rollback();
            MessageBox.Show("Šifra skupine že obstaja ali pa napaka pri
shranjevanju!");
        }
        //Osvežimo podatke v dataGridView1 glede na novo stanje bazi
        string SQL = "SELECT * FROM Drzave";
        dataConnection = new SqlConnection(izvor);
        daDrzave = new SqlDataAdapter(SQL, dataConnection);
        dsDrzave = new DataSet();
        daDrzave.Fill(dsDrzave, "Drzava");
        dataGridView1.DataSource = dsDrzave.Tables[0];
    }
}
```

Uporabniku želimo ponuditi še možnost iskanja oziroma preverjanja, ali je neka država že v tabeli. V odzivni metodi bomo preverjanje izvedli kar v gradniku *DataGridView*. Če najdemo iskano državo, si zapomnimo njen indeks. Preko tega indeksa lahko nato na koncu pridemo do vseh podatkov iskane države.

```
//ISKANJE države, katere ime je zapisano v textBox1
private void toolStripButton5_Click(object sender, EventArgs e)
{
    /*v spremenljivko pozicija bomo shranili indeks najdene države – če
      države ne najdemo, ima spremenljivka vrednost -1*/
    int pozicija=-1;
    for (int i = 0; i < dataGridView1.Rows.Count; i++)
    {
        if (Convert.ToString(dataGridView1.Rows[i].Cells[1].Value) ==
toolStripTextBox1.Text)
```

```
        {
            pozicija = i;
            break;
        }
    }
    if (pozicija == -1)
        MessageBox.Show("Ta država ne obstaja!");
    else
    {
        //v gradniku DataGridView se postavimo na najdeno državo
        dataGridView1.Rows[pozicija].Selected = true;
        /*če hočemo, da bo izbrana vrstica tudi aktivna, določimo še
          aktivno celico te vrstice*/
        dataGridView1.CurrentCell = dataGridView1.Rows[pozicija].Cells[0];
    }
}
```

Naslednja odzivna metoda je prirejena dogodku *Enter*, ki se zgodi ob vstopu uporabnikove miške v gradnik *TextBox*. Dosedanja vsebina vnosnega polja se ob vsakem vstopu briše.

```
//BRISANJE vsebine gradnika textBox1
private void toolStripTextBox1_Enter(object sender, EventArgs e)
{
    toolStripTextBox1.Clear();
}
```

Na vrsti so odzivne metode razreda *Drzava*. Objekte tega razreda bomo, kot smo že videli, potrebovali za vnašanje novih držav, ter za spreminjanje podatkov obstoječih držav. Obrazec smo priravili že prej, napisati morame le še potrebne odzivne metode.

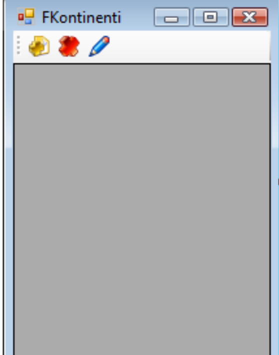
V konstruktorju poskrbimo za prenos vsebine tabele *Kontinenti* v gradnik tipa *ComboBox*, ki je na obrazcu. Nekaj podobnega smo delali že v eni od vaj, ne smemo pa pozabiti na lastnosti *DisplayMember* in *ValueMember*. S prvo povemo, kateri podatki bodo v spustnem seznamu prikazani, z drugo pa kateri podatek bo vrnjen ob izbiri. Dodali bomo že odzivno metodo dogodka *KeyPress*, ki jo bomo priredili vnosnim poljem. Pred zapirajem obrazca pa bomo še preverili, če je uporabnik korektno izpolnil vsa vnosna polja in izbral kontinent.

```
//konstruktor odzivne metode razreda Drzava
public partial class Drzava : Form
{
    public Drzava()    //Konstruktor
    {
        InitializeComponent();
        /*v gradnik ComboBox programsko uvozimo vse kontinente, ki se
          nahajajo v tabeli Kontinenti*/
        SqlConnection povezava = new SqlConnection();//Objekt za povezavo
        DataSet dsDrzaveSQL; //DataSet za shranjevanje prenešenih podatkov
        SqlDataAdapter daKontinenti;//adapter za prenos dobaviteljev
        try
        {
            povezava = new SqlConnection(Program.drzave.izvor);
```

```
            SqlCommand poizvedba = new SqlCommand();
            poizvedba.CommandText = "SELECT * FROM Kontinenti";
            poizvedba.Connection = povezava;
            daKontinenti = new SqlDataAdapter(poizvedba.CommandText,
                                              povezava);
            dsDrzaveSQL = new DataSet();
            //objekt dsDrzaveSQL povežemo s tabelo Kontinenti
            daKontinenti.Fill(dsDrzaveSQL,"Kontinenti");
            //objekt comboBox1 povežemo s podatkovnim izvorom
            comboBox1.DataSource = dsDrzaveSQL.Tables[0];
            //prikazana bodo imena Kontinentov
            comboBox1.DisplayMember = "Ime";
            /*ob izbiri kontinenta bo vrednost izbire ID tega kontinenta*/
            comboBox1.ValueMember = "ID_Kontinenta";
            //Odpremo povezavo z bazo, da se izvede poizvedba
            poizvedba.Connection.Open();
            povezava.Close();//Zapremo povezavo z bazo
        }
        catch
        {
            MessageBox.Show("Napaka pri dostopu do baze!\nProjekt se bo
                             zaprl!");
            Application.Exit();
        }
    }
    //dogodek KeyPress gradnika texBox2
    private void textBox2_KeyPress(object sender, KeyPressEventArgs e)
    {
        //dovoljen le vnos celega števila (števke in tipka BackSpace)
        if (((e.KeyChar < '0') || (e.KeyChar > '9'))&& (e.KeyChar !=
          (char)(8)))
            e.Handled = true;
    }
    //Dogodek gumba Shrani
    private void button1_Click(object sender, EventArgs e)
    {
        //Če uporabnik ne bo vnesel števila prebivalcev, se okno ne bo zaprlo
        if (textBox2.Text.Trim() == "")
        {
            MessageBox.Show("Vnesi število prebivalcev!");
            this.DialogResult = DialogResult.None;
        }
        //Če uporabnik ne bo vnesel površine, se okno ne bo zaprlo
        else if (textBox3.Text.Trim() == "")
        {
            MessageBox.Show("Vnesi površino!");
            this.DialogResult = DialogResult.None;
        }
        //Če uporabnik ne bo vnesel BDP, se okno ne bo zaprlo
        else if (textBox4.Text.Trim() == "")
        {
            MessageBox.Show("Vnesi BDP!");
            this.DialogResult = DialogResult.None;
```

```
        }
    }
}
```

Za ogled kontinentov, ki obstajajo v tabeli *Kontinenti* prav tako potrebujemo svoj obrazec, ki ga poimenujmo *FKontinenti.cs*. Vsebuje naj menijsko vrstico s tremi gumbi za dodajanje novega kontinenta, brisanje izbranega kontinenta in urejanje izbranega kontinenta. Podatki iz tabele bodo zopet prikazani v gradniku tipa *DataGridView*, ki mu lastnost *Dock* nastavimo na *Fill*.



Slika 147: Obrazec za prikaz, urejanje, brisanje in dodajanje kontinentov.

Obrazec bomo odpirali iz glavnega obrazca, zato imamo v glavnem meniju le-tega že pripravljeno postavko *Kontinenti*. Tule je odzivna metoda:

```
//PRIKAZ kontinentov
private void kontinentiToolStripMenuItem_Click(object sender, EventArgs e)
{
    FKontinenti Kont = new FKontinenti();
    Kont.ShowDialog();
}
```

Na obrazcu *FKontinenti* ni nobenega modalnega gumba, zato smo metodo *ShowDialog* klicali kar samostojno.

Kar nekaj dela imamo še z odzivnimi metodami tega obrazca. Kontinente želimo tudi dodajati, spreminjati njihove podatke in jih brisati iz tabele. V konstruktorju najprej poskrbimo za prenos vsebine tabele *Kontinenti* iz baze *Drzave*.

```
public partial class FKontinenti : Form
{
    SqlConnection dataConnection;
    SqlDataAdapter daKontinenti;
    DataSet dsDrzave; //DataSet za shranjevanje prenešenih tabel
    public FKontinenti()
    {
        InitializeComponent();
        //Klic metode za prikaz podatkov in oblik. gradnika DataGridView
        OblikujDataGridView();
    }
    //metoda za prikaz podatkov v dataGridView1 in za njegovo oblikovanje
    private void OblikujDataGridView()
    {
        string SQL = "select * from Kontinenti";
        dataConnection = new SqlConnection(Program.drzave.izvor);
```

```
        dsDrzave = new DataSet();
        daKontinenti = new SqlDataAdapter(SQL, dataConnection);
        daKontinenti.Fill(dsDrzave, "Kontinenti");
        dataGridView1.DataSource = "";
        /*iz dsDrzave prikažemo tabelo z indeksom 0 - Kontinetni (v tabeli
          z indeksom 0 so države*/
        dataGridView1.DataSource = dsDrzave.Tables[0];
        //prvi stolpec (ID kontinenta) v dataGridView1 ne bo prikazan
        dataGridView1.Columns[0].Visible = false;
        dataGridView1.Columns[1].Width = 150;
        dataGridView1.AllowUserToAddRows= false;//dodajanje vrstic ni možno
         //brisanje vrstic ni možno
        dataGridView1.AllowUserToDeleteRows= false;
        //urejanje neposredno na obrazcu NI možno
        this.dataGridView1.ReadOnly = true;
        dataGridView1.Rows[0].Selected = true;
        /*če hočemo, da bo izbrana vrstica tudi aktivna, določimo še
         aktivno celico te vrstice*/
        dataGridView1.CurrentCell = dataGridView1.Rows[0].Cells[1];
        dataConnection.Close();
    }
}
```

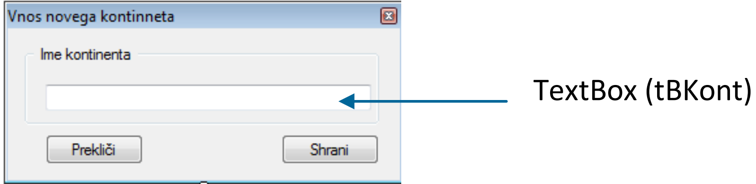
Odzivne metode za brisanje, dodajanje in urejanje podatkov iz tabele *Kontinenti* so zelo podobne tistim, ki smo jih uporabili pri delu s tabelo *Drzava*. Za dodajanje in ažuriranje bomo potrebovali tudi nov obrazec, na katerem bo eno samo vnosno polje in dva modalna gumba. Tule so najprej vse tri odzivne metode.

```
    //BRISANJE izbranega kontinneta
    private void toolStripButton2_Click(object sender, EventArgs e)
    {
        if (MessageBox.Show("Brišem kontinent
"+dataGridView1.CurrentRow.Cells[1].Value + "?","BRISANJE
POSTAVKE",MessageBoxButtons.YesNo,MessageBoxIcon.Question)==DialogResult.Yes)
        {
            /*POZOR: pred brisanjem kontinenta bi morali še preveriti, ali
             v tabeli Drzave obstaja kaka država iz tega kontinenta in v
             tem primeru brisanje preprečiti!!!*/
            int sifra;
            sifra=Convert.ToInt32(dataGridView1.CurrentRow.Cells[0]. Value);
            dataConnection.Open();
            SqlTransaction myTrans = dataConnection.BeginTransaction();
            SqlCommand dataCommand = new SqlCommand();
            dataCommand.Connection = dataConnection;
            dataCommand.Transaction = myTrans;
            try
            {
                dataCommand.Parameters.Add(new SqlParameter("@ID_Kont",
                                           sifra));
                dataCommand.CommandText = "DELETE FROM Kontinenti WHERE
                                           ID_Kontinenta =@ID_Kont";
                dataCommand.ExecuteNonQuery();
                myTrans.Commit();
```

```
            }
            catch
            {
                //Če je prišlo do napake, vzpostavimo prejšnje stanje
                myTrans.Rollback();
                MessageBox.Show("Napaka pri shranjevanju!");
            }
            dataConnection.Close();
            OblikujDataGridView();
        }
    }
    //DODAJANJE novega kontinenta
    private void toolStripButton1_Click(object sender, EventArgs e)
    {
        FNovKontinent novK = new FNovKontinent();
        if (novK.ShowDialog() == DialogResult.OK)
        {
            //Pred začetkom transakcije odpremo povezavo z bazo podatkov
            dataConnection.Open();
            SqlTransaction myTrans = dataConnection.BeginTransaction();
            SqlCommand dataCommand = new SqlCommand();
            dataCommand.Connection = dataConnection;
            dataCommand.Transaction = myTrans;
            try
            {
                //deklaracija parametrov za SQL transacijo
                dataCommand.Parameters.Add(new SqlParameter("@Ime",
                                           novK.tBKont.Text));
                dataCommand.CommandText = "Insert INTO Kontinenti (ime)
                                           VALUES (@Ime)";
                dataCommand.CommandType = CommandType.Text;
                dataCommand.ExecuteNonQuery();
                myTrans.Commit();
            }
            catch
            {
                //Če je prišlo do napake, vzpostavimo prejšnje stanje
                myTrans.Rollback();
                MessageBox.Show("Napaka pri shranjevanju!");
            }
            dataConnection.Close();
            OblikujDataGridView();
        }
    }
    //UREJANJE izbranega kontinenta
    private void toolStripButton3_Click(object sender, EventArgs e)
    {
        FNovKontinent novK = new FNovKontinent();
        //POZOR:tBKont mora imeti lastnost Modifiers nastavljeno na Public
       novK.tBKont.Text = dataGridView1.CurrentRow.Cells[1].Value.ToString();
        /*POZOR: gradnik groupBox1 mora imeti lastnost Modifiers
          nastavljeno na Public*/
        novK.groupBox1.Text = "Ažuriranje imena kontinenta";
```

```
        if (novK.ShowDialog() == DialogResult.OK)
        {
            //Pred začetkom transakcije odpremo povezavo z bazo podatkov
            dataConnection.Open();
            SqlTransaction myTrans = dataConnection.BeginTransaction();
            SqlCommand dataCommand = new SqlCommand();
            dataCommand.Connection = dataConnection;
            dataCommand.Transaction = myTrans;
            try
            {
                //deklaracija parametrov za SQL transacijo
                dataCommand.Parameters.Add(new SqlParameter("@Ime",
                  novK.tBKont.Text));
                dataCommand.Parameters.Add(new SqlParameter("@Kont",
                  Convert.ToInt32(dataGridView1.CurrentRow.Cells[0].Value)));
                /*spremenimo le ime kotinenta, ID ostane seveda enak
                  (ključno polje)*/
                dataCommand.CommandText = "Update Kontinenti SET Ime=@Ime
                                           WHERE ID_Kontinenta=@Kont";
                dataCommand.CommandType = CommandType.Text;
                dataCommand.ExecuteNonQuery();
                myTrans.Commit();
            }
            catch
            {
                myTrans.Rollback();
                MessageBox.Show("Napaka pri shranjevanju!");
            }
            dataConnection.Close();
            OblikujDataGridView();
        }
    }
}
```

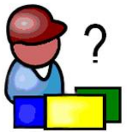
Ostane nam še obrazec za dodajanje oz. urejanje kontinenta(*FNovKontinent.cs*). Gumba na obrazcu sta modalna: gumb *Shrani* vrne vrednost *DialogResult.OK*, gumb *Prekliči* pa *DialogResult.Cancel*.



Slika 148: Obrazec dodajanje/ažuriranje kontinenta.

```
public partial class FNovKontinent : Form
{
    public FNovKontinent()
    {
        InitializeComponent();
        /*gradnika tBKont in groupBox morata imeti lastnost Modifiers
```

```
          nastavljeno na Public*/
    }
}
```



PRIPRAVA NAMESTITVENEGA PROGRAMA, TESTIRANJE

Na neki točki v procesu razvoja *Windows* aplikacije, je potrebno projekt pripraviti za testiranje na ciljnem računalniku. Pripraviti ga je potrebno tudi tako, da ga bodo uporabniki lahko namestili na svoj računalnik in ga tudi uporabljali. Testiranje projekta pri uporabniku nam lahko v veliki meri pomaga pri iskanju boljših rešitev in pri odpravljanju težav, do katerih lahko pride pri uporabi aplikacije na drugem računalniku.

V preteklosti je bilo večina projektov nemščenih s pomočjo raznih namestitvenih (***Setup***) programov, ki so bili shranjeni na *CD*-ju ali nekje v omrežni soseščini. V zadnjih letih pa lahko *Windows* projekte namestimo neposredno preko *Web* strežnikov, kar je posebno ugodno pri nadgradnji projektov. Kot primer lahko vzamemo programe za protivirusno zaščito, ki jih je zaradi stalno novih in novih virusov potrebno nadgrajevati zelo pogosto. *Visual Studio 2010* nam za potrebe testiranja, namestitev in nadgradenj ponuja oba načina.

Obstajajo trije načini za razmnoževanje oz. testiranje aplikacij.

- *XCopy*
- *ClickOnce*
- Izdelava *Setup* projekta– a (namestitvenega programa)

Vsak od teh načinov ima svoje prednosti, pa tudi slabosti.

XCopy

Najstarejši način za prenos aplikacije na drug sistem je s pomočjo metode *xcopy* operacijskega sistema. Mapo, v katero je shranjena naša izvršilna datoteka (.*exe*) na sistemu, na katerem smo ga razvili, enostavno prekopiramo na uporabnikov disk. Če so v tej mapi vse potrebne datoteke za našo aplikacijo, se bo projekt pravilno odprl in uporaba aplikacije oziroma njeno testiranje bo možno tudi na tem računalniku. Prednost tega načina je, da ni potrebna nobena dodatna konfiguracija ali pa registracija produkta.

Postopek je naslednji : končna izvršilna verzija projekta, ki smo ga razvili, je na našem sistemu shranjena v podmapi našega projekta in sicer v mapi *…bin\Release*. S pomočjo ukaza za

kopiranje datotek ***XCopy*** operacijskega sistema *DOS* celotno mapo prekopiramo na nek medij (npr. CD, ključ, medij v omrežni soseščini …).

Primer ukaza *XCopy*

C:>\ xcopy "C:\Moja aplikacija\bin\release" "E:\Moja aplikacija\" /s*

Seveda lahko kopiranje izvedemo tudi s pomočjo tehnike *Copy – Paste v Windows Explorerju*.

Ta način je seveda najenostavnejši, ima pa veliko pomanjkljivosti. Če namreč v mapi *Release* ni vseh potrebnih datotek, ali pa če ciljni uporabnik nima nameščenega okolja .NET Framework, aplikacije na njegovem računalniku ne bo možno zagnati.

ClickOnce

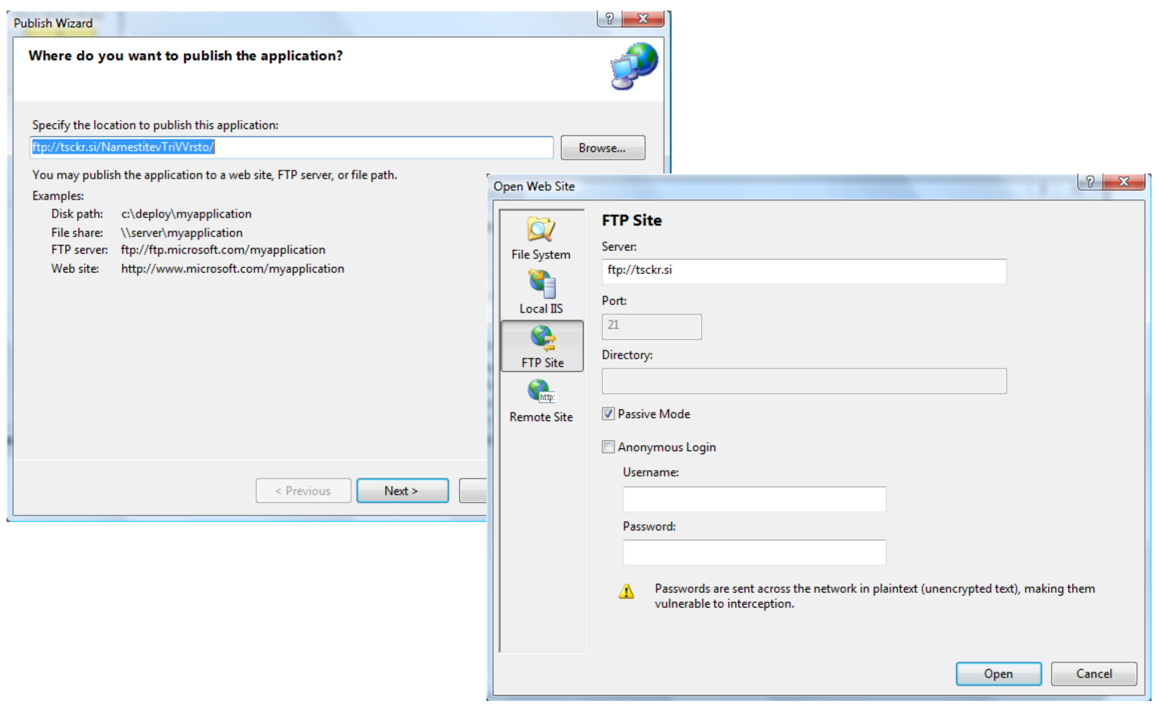
Pri izdelavi namestitvenega projekta, ki bo na voljo uporabnikom preko spletnega strežnika, ali pa tudi preko klasične metode (npr. instalacijski CD ipd.) nam pomaga poseben čarovnik (*Wizard*), vsebovan v razvojnem okolju *Visual C#*.

Izberimo projekt, za katerega bomo naredili nov namestitveni projekt in ga postavili na nek strežnik, kjer bo na voljo uporabnikom. Kot alternativo bomo ustvarili še namestitveni program, s pomočjo katerega bo možno naš projekt namestiti pri uporabniku kar preko ključa oz. CD-ja. Izberimo npr. projekt *TriVVrsto*, ki smo ga izdelali v enem od prejšnjih poglavij.

POZOR! V primeru, da bomo namestitvene datoteke ustvarili na nekem strežniku, bo na strežniku v izbrani mapi poleg namestitvenih datotek nastala tudi datoteka Publish.htm, ki omogoča instalacijo preko spleta (npr. s pomočjo *Microsoft Internet Explorer*-ja). Če pa bomo namestitvene datoteke ustvarili na nekem disku, bomo na le-tem dobili le namestitvene datoteke (najpomembnejša med njimi je *Setup.exe*, ki požene namestitev).

Odprimo torej najprej projekt *TriVVrsto*. Z desnim klikom na projekt v oknu *Solution Explorer* poženemo čarovnika za izdelavo namestitvene aplikacije. Odpre se novo okno, v katerem izberemo postavko *Publish*, da se odpre *Publish Wizard*.

V prvem koraku moramo navesti lokacijo, na kateri želimo objaviti (oz. kamor želimo postaviti) naše namestitvene datoteke. Na voljo imamo več možnosti. Namestitveno datoteko, ali pa več datotek, lahko namreč ustvarimo na strežniku, kjerkoli na disku, na *ftp* strežniku ali pa na spletni strani. Možnosti se odprejo s klikom na gumb *Browse*. V našem primeru bomo namestitveno datoteko ustvarili na *ftp* strežniku *TŠC Kranj* ([*ftp://tsckr.si*](ftp://tsckr.si)), seveda pa moramo pred tem v obrazec *Open Web Site* vnesti svoje uporabniško ime in geslo. Ciljno mapo na *ftp* strežniku ustvari namestitveni program sam!

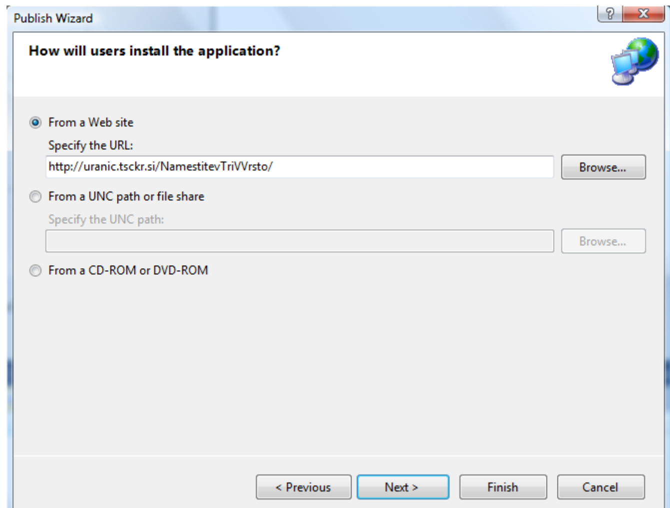


Slika 149: Ustvarjanje namestitvenih datotek na *ftp* strežniku.

Če bi hoteli namestitvene datoteke ustvariti na nekem lokalnem strežniku, bi v oknu *Open Web Site* izbrali *File System* in izbrali namestitveno mapo na nekem strežniku.

V drugem koraku določimo, kako bo uporabnik instaliral aplikacijo. Izbrati moramo eno od treh ponujenih opcij

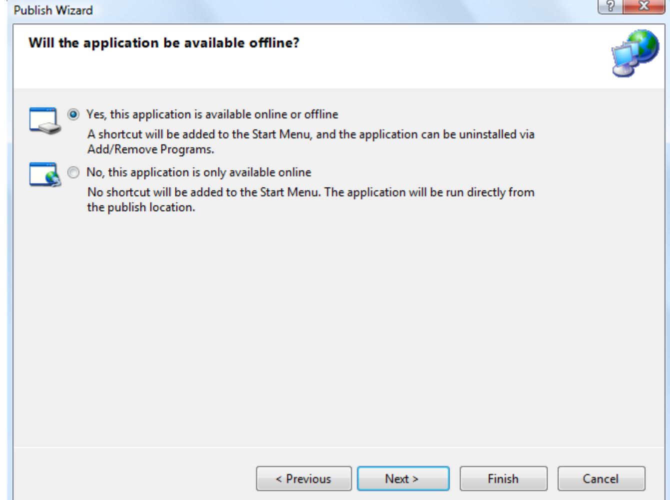
- *From a Web site* - instalacija z neke spletne strani .
- *From a UNC path or file share* – instalacija preko strežnika v *Windows* okolju (mreži).
- *From a CD-ROM or DVD-ROM* –instalacija s *CD*-ja oz *DVD*-ja.



V našem primeru izberimo prvo možnost: potrebno je še zapisati naslov strežnika in mape, v katero se bodo shranile namestitvene datoteke.

Slika 150: Okno v katerem določimo, kako bo uporabnik namestil svojo aplikacijo.

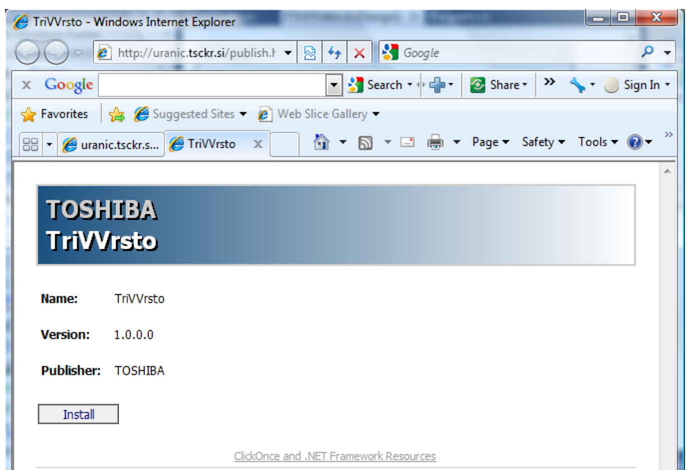
V tretjem koraku določimo, ali bo naša aplikacija dostopna *offline*. Privzeta je vrednost *Yes*, kar za našo trenutno namestitev pomeni, da jo uporabnik lahko namesti brez vnaprejšnje prijave na nek strežnik oz. spletno stran.



Slika 151: Okno v katerem določimo, ali bo za namestitev potrebna prijava ali ne!

V četrtem koraku imamo možnost potrditve vseh namestitev. S klikom na gumb *Finish* bo čarovnik na strežniku ustvaril ustrezno mapo, v njej pa vse potrebne datoteke za namestitev, hkrati pa še ustrezen certifikat, ki ga lahko uporabljamo za testiranje. Obenem se spremenijo/dodajo nekatere nastavitve (datoteke) v *Solution Explorer*-ju, to datoteke potrebne za zagotovitev prijavljanja, varnosti in razmnoževanja aplikacije. Za kasnejšni pregled in spreminjanje teh nastavitev lahko kadarkoli v *Solution Explorer*-ju izberemo projektno datoteko, kliknemo desni miškin gumb in izberemo *Properties*. V oknu ki se prikaže izberemo ustrezen zavihek (npr. zavihek o varnosti – *Security*) in opravimo želene spremembe.

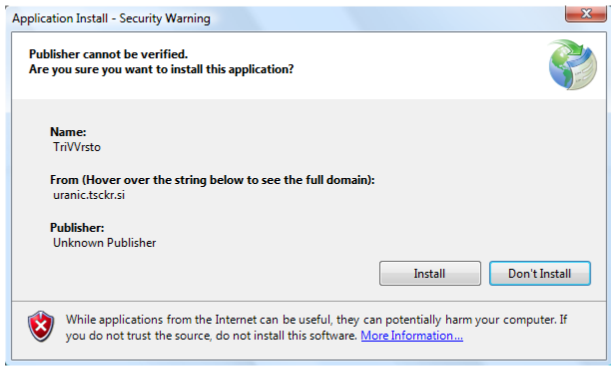
Namestitvene datoteke so sedaj pripravljene. Če smo v prvem koraku izbrali strežnik, *ftp* strežnik oz. spletno stran, je ob ustvarjanju namestitvenih datotek nastala tudi datoteka *Publish.htm*, ki omogoča namestitev preko spleta (npr. s pomočjo *Microsoft Internet Explorerja*), torej *namestitev aplikacije preko spletne strani*. Bodočim uporabnikom naše aplikacije sedaj le pošljemo ustrezen *link* za dostop do namestitve.



Slika 152: Namestitveno okno naše aplikacije.

Namestitev izvedemo tako, da se v internet brskalniku postavimo v mapo na spletnem strežniku in kliknemo datoteko *Publish.htm*, da se odpre okno za namestitev. Prikaže se namestitveno okno, v njem pa glavne lastnosti projekta, ki ga želimo namestiti.

Namestitev zaženemo s klikom na gumb *Install*. Na ekranu se prikaže pogovorno okno za namestitev nove aplikacije.



Slika 153: Varnostno opozorilo pred namestitvijo!

V splošnem pogovorno okno vsebuje tudi varnostno opozorilo, ker pač naša testna aplikacija vsebuje le testni certifikat, ne pa tudi certifikat s pooblastilom. Ne glede na to obvestilo bodo naši uporabniki vedeli, da aplikacija prihaja z mesta, ki mu lahko zaupajo, še posebno kadar gre za namestitev preko lokalnega intraneta. O tem seveda lahko uporabnike prej tudi obvestimo s posebnim sporočilom preko elektronske pošte. Varnostnega opozorila pa se seveda lahko znebimo z namestitvijo veljavnega certifikata.

Po namestitvi bo v meni *Start* dodana bližnjica za zagon aplikacije, možna pa bo tudi odstranitev te aplikacije preko opcije *Dodaj ali odstrani programe* v *Nadzorni plošči (Control Panel)*.

Možna je seveda tudi izdelava aplikacije, ki je dostopna le *online*. V tem primeru se aplikacija na uporabnikovem računalniku požene neposredno s strežnika. V tem primeru ni potrebno ustvarjanje bližnjice v *Start* meniju in obenem tudi ne zmožnost odstranitve aplikacije. Potrebna je le vsakokratna vzpostavitev povezave s strežnikom in *download* aplikacije preko tega strežnika pred njeno uporabo, kar pa lahko traja dalj časa.

Za vsako *ClickOnce* namestitev je možno tudi avtomatsko posodabljanje oz. *update*, če je vzpostavljena povezava z ustreznim strežnikom, na katerem je projekt ustvarjen. Recimo, da na strežniku obstaja novejša verzija aplikacije *TriVVrsto*. Ko uporabnik starta svojo verzijo aplikacije, le ta najprej preveri, ali na strežniku obstaja novejša različica. Če obstaja novejša verzija, se odpre pogovorno okno s sporočilom, da obstaja novejša različica ter nam ponudi *download* te nove verzije. S klikom na gumb *OK* pogovornega okna se bo nova različica namestila in se odslej naprej tudi izvajala na uporabnikovem računalniku (vse do naslednje nove verzije).

Kdaj in v katerem primeru se bo prikazalo okno z obvestilom in nastavitvami za nadgradnje naše aplikacije, pa lahko določimo v pogovornem oknu *Application Updates*. Okno odpremo tako, da v *Solution* Explorer-ju izberemo projekt, kliknemo desni miškin gumb, izberemo *Properties.* V

oknu, ki se prikaže, izberemo zavihek *Publish* in kliknemo na gumb *Updates*. Odpre se pogovorno okno za določanje oz. spreminjanje privzetih lastnosti za posodobitve. Če na primer želimo, da se naša aplikacija zažene preden so naložene posodobitve, izberemo prvi radijski gumb. Tako se bo najnovejša verzija aplikacije zagnala šele, ko jo bomo odprli naslednjič. V tem primeru imamo možnost še dodatne nastavitve, saj lahko določimo tudi interval, kako pogosto naj naša aplikacija preverja, ali obstajajo posodobitve. Na vrhu tega sporočilnega okna obstaja tudi *Potrditveni gumb (CheckBox)* s pomočjo katerega lahko izključimo preverjanje obstoja novih verzij oziroma posodobitev.

Na dnu okna je še gumb za določanje minimalne zahtevane verzije programa, ki jo je še možno posodobiti, ter možnost izbire druge lokacije od koder se bo izvedla posodobitev.

Setup (namestitveni) program

Če nam prva dva načina za izdelavo novih verzij programov ne zadoščata, je na voljo še tretja, najbolj kompleksna. Žal pa ta verzija ni na voljo v *Visual C# Express Edition*, ampak le v višjih različicah razvojnega orodja ***Visual C#*** (npr. *Standard Edition*).

DOKUMENTIRANJE

Dokumentiranje razredov

Kot že vemo, lahko splošne komentarje v našo kodo dodajamo tako, da stavke s komentarjem začnemo z znakoma **//** za enovrstični komentar ali pa s parom znakov /* oz */ za večvrstični komentar.

Pri pisanju novih razredov in njihovih članov pa lahko uporabljamo tudi XML dokumentacijo. XML dokumentacija bo naše razrede naredila razumljivejše za ostale razvijalce, ki bodo mogoče nadaljevali z našim projektom, ali pa samo uporabljali razred, ki smo ga napisali. Tudi za nas je takšno pisanje dokumentacije koristno, saj če imamo veliko lastnih razredov, bomo s časoma pozabili njihov namen in možnosti uporabe. S pomočjo *XML* komentarjev pa bomo njihovo uporabo in namen hitro obnovili. Informacija, ki jo bomo napisali kot *XML* dokumentacijo, se bo tako ob uporabi našega razreda in ustrezne kombinacije tipk prikazala na ekranu in sicer v okvirčku pod imenom razreda ali metode.

Čeprav *XML* dokumentacija temelji na *XML* sintaksi, nam ni potrebno podrobno poznavanje *XML*-a da bi ga lahko uporabljali. Vedeti moramo le, da se vrstica z *XML* dokumentacijo prične s tremi znaki /// (tremi *slash*-i), nahajati pa se morajo takoj za razredom oz. njegovim članom, ki ga želimo dokumentirati.

Dokumentacija za razred ali člana razreda lahko vsebuje enega ali več elementov dokumentiranja. Vsak element se prične z oznako za začetek, npr. *<summary>* in se konča z oznako za konec elementa, npr. *</summary>*. Vsebina tega elementa dokumentacije se nahaja med začetno in končno oznako.

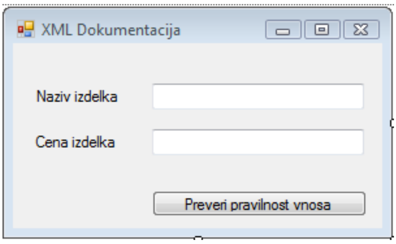
XML	Razlaga
<summary>	Zagotavljanje temeljne razlage o nekem razredu, lastnosti, metodi ali drugem elementu.
<value>	Uporablja se za opis oz. razlago vrednosti neke lastnosti.
<returns>	Uporablja se za opis oz. razlago vrnjene vrednosti neke metode.
<param name>="name**"**	Uporablja se za opis oz. razlago parametrov metode.

Tabela 25: Tabela najpogostejših elementov XML dokumentacije.

Zapisano je že bilo, da je najlažji način za ustvarjanje *XML* dokumentacije tak, da v prvi vrstici, ki sledi najavi novega razreda, vtipkamo trojno poševnico oz. *slash* (///). *Visual Studio* nam nato samodejno naredi ogrodje *XML* dokumentacije, vključno s tem, ali je oznaka primerna za člana

razreda, ki ga dokumentiramo. Če npr. vtipkamo *///* v prazni vrstici pred članom razreda, ki ima parametre in vrača nek rezultat, bo *Visual Studio* tudi avtomatično generiral oznako *<summary>* za to metodo, oznako *<param name =…>* za posamezne parametre te metode in oznako *<return>* za vrednost, ki jo metoda vrača. Nato lahko med oznakami za začetek in konec nekega elementa *XML* dokumentacije pričnemo s tipkanjem opisa posamezne sekcije.

Projekt *XML Dokumentacija* demonstrira primer *XML* dokumentacije razredov. Na obrazcu sta le dve vnosni polji: v prvo naj bi uporabnik vnesel poljuben tekst, ki pa ne sme biti prazen, v drugo polje pa naj bi vnesel neko decimalno število. Po kliku na gumb *Preveri pravilnost vnosa* bomo s pomočjo dveh statičnih metod razreda *Validator* preverili pravilnost uporabnikovega vnosa. Razred *Validator* smo napisali sami in ga opremili z *XML* dokumentacijo.



Slika 154: Projekt *XMLDokumentacija*.

```
public partial class FXML : Form
{
    public FXML()
    {
        InitializeComponent();
    }
    /// <summary>
    /// Vsebuje statične metode za kontrolo pravilnosti uporabnikovega vnosa
    /// </summary>
    public class Validator
    {
        public Validator()//konstruktor
        {
        }

        /// <summary>
        /// Besedilo, ki se bo izpisalo NA pogovornem oknu ob napačnem vnosu
        /// </summary>
        public static string Naslov="Napaka pri vnosu!";

        //statična metoda, zato jo lahko kličeno kar s pomočjo razreda
        /// <summary>
        /// Preverja, če je uporabnik vnesel podatke v tekstovno polje
        /// </summary>
        /// <param name="textBox">Tekstovno polje, ki se preverja</param>
        /// <returns>True (v redu), če je uporabnik vnesel podatke</returns>
        public static bool IsPresent(TextBox textBox)
        {
            if (textBox.Text=="")
            {
```

```
                /*Predpostavimo, da je pomen vnosnega polja zapisan v
                  lastnosti Tag*/
                MessageBox.Show("Polje "+textBox.Tag+" ne sme biti
prazno!",Naslov);
                textBox.Focus();
                return false;
            }
            return true;
        }
        //statična metoda, zato jo lahko kličeno kar s pomočjo razreda
        public static bool jeDecimalno(TextBox textBox)
        {
            try
            {
                Convert.ToDecimal(textBox.Text);
                return true;
            }
            catch
            {
                /*Predpostavimo, da je pomen vnosnega polja zapisan v
                  lastnosti Tag*/
                MessageBox.Show(textBox.Tag + " mora biti decimalno
število.", Naslov);
                textBox.Focus();
                return false;
            }
        }
    }
    private void bPreveri_Click(object sender, EventArgs e)
    {
        if (Validator.IsPresent(tBNaziv) && Validator.jeDecimalno(tBCena))
            MessageBox.Show("Vnos pravilen!");
    }
}
```

Ko smo *XML* dokumentacijo dodali v naš razred, jo bo *Visual Studio* uporabil npr. tedaj, ko bo uporabnik zapisal ime razreda, ali pa ime metode, pa tudi tedaj, ko se le z miško zapelje prek kode.

V prvem primeru, ko vtipkamo ime razreda *Validator* in za njim še znak pika, *IntelliSense* prikaže seznam članov razreda *Validator*. Med njimi je tudi metoda *IsPresent*, ki smo jo napisali. Ob njej se v okvirčku pojavi okno z opisano dokumentacijo za ta razred, ki smo jo napisali s pomočjo *XML*-a.

```
private void bPreveri_Click(object sender, EventArgs e)
{
    if (Validator.
}
          Equals
          IsPresent        bool Validator.IsPresent(TextBox textBox)
                           Preverja, če je uporabnik vnesel podatke v tekstovno polje
          jeDecimalno
          Naslov
          ReferenceEquals
```

Slika 155: Sistem *IntelliSense* poleg imena metode prikaže tudi okno z našo dokumentacijo.

Ko uporabnik izbere metodo *IsPresent* in za njo zapiše oklepaj, *Visual Studio* v okvirčku pod metodo izpiše razlago za parameter te metode, tako kot smo jo napisali s pomočjo *XML* dokumentacije.

```
private void bPreveri_Click(object sender, EventArgs e)
{
    if (Validator.IsPresent(
}        bool Validator.IsPresent(TextBox textBox)
         textBox:
           Tekstovno polje, ki se preverja
```

Slika 156: Izpis *XML* dokumentacije za parameter metode.

Informacijo o samem razredu pa lahko uporabnik dobi, če se s kazalnikom miške postavi kamorkoli na besedico *Validator* (ime razreda) v kodi.

```
private void bPreveri_Click(object sender, EventArgs e)
{
    if (Validator.IsPresent(tBNaziv) && Validator.jeDecimalno(tBCena))
         class XMLDokumentacija.FXML.Validator
}        Vsebuje statične metode za kontrolo pravilnosti uporabnikovega vnosa
```

Slika 157: Izpis dokumentacije o našem razredu.

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