UNIVERZA V LJUBLJANI

FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – praktična matematika (VSŠ)

Uroš Makarić

Uporaba okolja Visual Studio Community za razvoj programov z grafičnim uporabniškim vmesnikom

Diplomska naloga (visokošolski strokovni študij)

LJUBLJANA, 2016

Zahvala

Za potrpežljivost pri pregledovanju ter vodenje in usmerjanje pri izdelavi diplomske naloge se zahvaljujem mentorju mag. Matiji Lokarju.

Prav posebna zahvala gre vsem mojim bližnjim, posebej staršem in dekletu, ki so mi ob nastajanju diplomske naloge ves čas stali ob strani in mi pomagali po svojih najboljših močeh.

Kazalo vsebine

PROGRAM DELA

V diplomski nalogi predstavite razvojno okolje Visual Studio Comunity s poudarkom na uporabi pri razvoju programov s programskim jezikom C#, ki zahtevajo grafični uporabniški vmesnik. Predstavite tako način dela z uporabo pristopa Windows Forms kot tudi s tehnologijo Windows Presentation Foundation. Opišite gradnike, ki jih pri takem razvoju najbolj pogosto uporabljamo.

Priporočena literatura:

- Perkins, B.; Hammer, J. V., Beginning C# 6 Programming with Visual Studio 2015, Wrox, 2015
- Whitaker, RB, The C# Player's Guide (2nd Edition), Starbound Software, 2015
- Sharp, J., Microsoft Visual C# Step by Step (8th Edition) (Developer Reference) 8th Edition, Microsoft Press, 2015

Ljubljana, april 2016 Mentor:

viš. pred. mag. Matija Lokar

Povzetek

V diplomski nalogi bomo spoznali različne gradnike, s katerimi razvijamo aplikacije v razvojnem okolju Visual Studio Community. Bolj natančno si bomo pogledali razvoj Windows Forms in WPF aplikacij. V nalogi bomo videli tudi določene razlike pri razvoju teh dveh vrst grafičnega uporabniškega vmesnika. Pri pisanju bomo domnevali, da bralec pozna osnove programskega jezika C#.
V prvem delu bomo bolj natančno opisali okolje Visual Studio Community in osnovna orodja tega okolja, s katerimi si poenostavimo delo pri razvoju. V drugem delu pa bomo spoznali gradnike, s katerimi razvijamo aplikacije z grafičnim uporabniškim vmesnikom.

Abstract

In this thesis, we will learn about various controls we use in developing applications in Visual Studio Community development environment. The development of Windows Forms and WPF applications will be examined in more detail, and some differences in the development of these two types of graphical user interface will be explained. It is assumed that the reader knows the basics of C# programming language.
In the first part, the Visual Studio Community environment will be described more accurately, as well as tools which simplify the development. In the second part, we will learn about controls which we use to develop applications with the designer graphical user interface.

Math. Subj. Class. (2010): 68N15, 68N25, 68U07, 68N20, 68N25
Computing Review Class. System (1998): D.2.2, D.2.3, D.2.5, D.2.6, D.3.2., D.3.3

KLJUČNE BESEDE: grafični uporabniški vmesnik, dogodkovno programiranje, integrirano razvojno okolje, Windows Forms, WPF, C#, Visual Studio 2015 Community, gradniki, XAML

KEY WORDS: graphical user interface, event-driven programming, integrated development environment, Windows Forms, WPF, C#, Visual Studio 2015 Community, controls, XAML

Uvod

Pri uporabi računalnika se večinoma srečujemo z grafičnimi uporabniškimi vmesniki (GUV), ki nam poenostavljajo komunikacijo z računalnikom. V diplomski nalogi bomo predstavili izdelavo GUV aplikacij v razvojnem okolju Visual Studio 2015 Community. Zadnjega bomo v nadaljevanju večkrat označili kar s kratico VS. Pri tem si bomo pogledali, kako si lahko pomagamo z orodji, ki jih imamo v razvojnem okolju in hitreje opravimo delo.

Vsi primeri aplikacij, ki jih bomo predstavili v nalogi, bodo zapisani v programskem jeziku C#. Ob tem bomo predpostavili, da bralec že pozna osnove programskega jezika C#, medtem ko razvojnega okolja VS 2015 Community še ne pozna, prav tako pa ne pozna osnovnih načel razvoja GUV aplikacij.

Z uporabo primerov bomo spoznali gradnike, s katerimi ustvarjamo GUV aplikacije. Ob tem bomo ugotovili, da je pred samim začetkom ustvarjanja nujno pripraviti idejni načrt aplikacije in ga nato postopoma uresničevati.

Visual Studio je razvojno okolje za ustvarjanje aplikacij, ki delujejo znotraj operacijskega sistema (OS) Windows. Spada med tako imenovana integrirana razvojna okolja (Integrated Development Environment – IDE, v nadaljevanju bomo uporabljali kratico IRO). IRO je programska oprema, kjer so na enem mestu zbrana različna orodja, ki jih potrebujejo razvijalci pri razvoju programske opreme. V vsakem IRO ima razvijalec na voljo urejevalnik kode, prevajalnik in razhroščevalnik. V urejevalnik kode je vgrajenih veliko zmožnosti, ki razvijalcu olajšajo delo, kot so npr. kontekstno barvanje kode, samodopolnjevanje, sprotna pomoč, označevanje sintaktičnih napak … Številna okolja pa ponujajo tudi druga orodja, ki jih utegne razvijalec potrebovati pri svojem delu. Seveda aplikacije lahko razvijamo s pomočjo urejevalnikov, kot je npr. Beležnica, jih prevajamo s klicanjem prevajalnikov z ukazno vrstico, vendar lahko večina razvijalcev bistveno hitreje in tudi lažje ustvarja v IRO.

Visual Studio je IRO, ki ga razvija podjetje Microsoft. S tem okoljem ustvarjamo spletne strani, programe, aplikacije, mobilne aplikacije, igre ..., ki jih izvajamo znotraj operacijskega sistema Windows.

V VS lahko razvijamo aplikacije v različnih programskih jezikih. Poleg jezika C#, ki ga bomo uporabljali v tej nalogi, lahko uporabljamo številne druge, kot so C++, Visual Basic, Python, F# … Znotraj razvojnega okolja VS imamo urejevalnik kode (Code Editor), oblikovalca (Designer), razhroščevalnik (Debugger) in druga orodja (Team Explorer, Solution Explorer, Properties Editor ...), s katerimi si poenostavimo delo ter pohitrimo razvoj aplikacij.

Poleg same kode moramo pri ustvarjanju aplikacij pogosto oblikovati in načrtovati videz določenih stvari. To so lahko tako videz uporabniškega vmesnika, shematski diagram baze podatkov in drugo. Visual Studio pozna več vrst orodij, s katerimi si pomagamo pri oblikovanju tovrstni stvari. Ta orodja poimenujemo s skupno besedo oblikovalec (Designer). VS ponuja več tovrstnih oblikovalcev. Tako na primer Windows Forms Designer uporabljamo za izdelavo GUV aplikacij, ki temeljijo na uporabi oken, kot jih poznamo v okolju Windows.

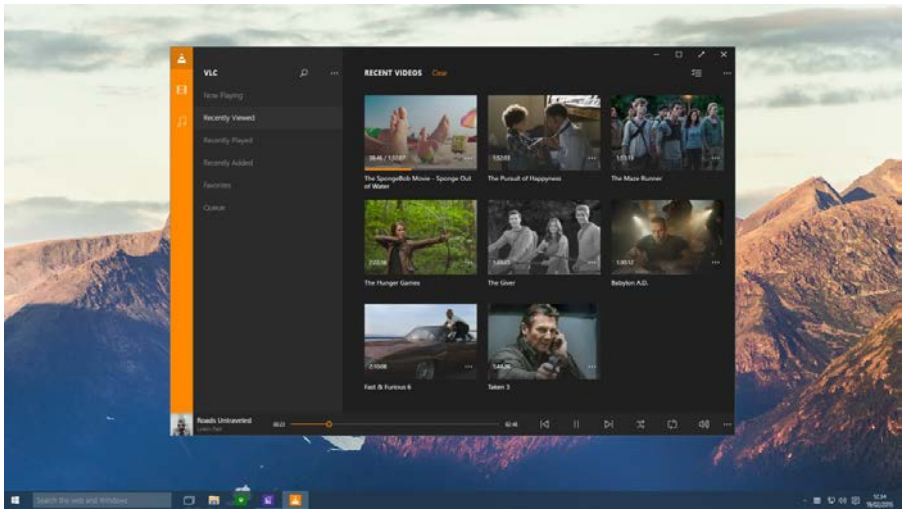
Grafični uporabniški vmesnik (Graphical User Interface – GUI) poenostavi uporabo programa tako, da izkoristi grafične zmožnosti računalnika oz. elektronskih naprav. Tako lahko praktično vsak uporablja računalnik tudi, če ne pozna programskega jezika. Dogodki v GUV aplikacijah se večinoma urejajo preko grafičnih elementov, kot je npr. kazalec miške. Z dobrim načrtovanjem GUV uporabniku ni treba razumeti le programskega jezika, temveč mu ni treba razumeti niti poteka navodil, da lahko uporablja aplikacijo. Vsi, ki smo uporabljali OS Microsoft Windows, smo se že seznanili z GUV. Pri uporabi se zanašamo na kazalec, s katerim se premikamo po zaslonu, za premik le-tega pa skrbi miška, ki je povezana z računalnikom. Zaradi grafičnega prikaza je uporabnikom lažje in razumljiveje uporabljati tovrstne programe.

Grafični uporabniški vmesnik (GUV)

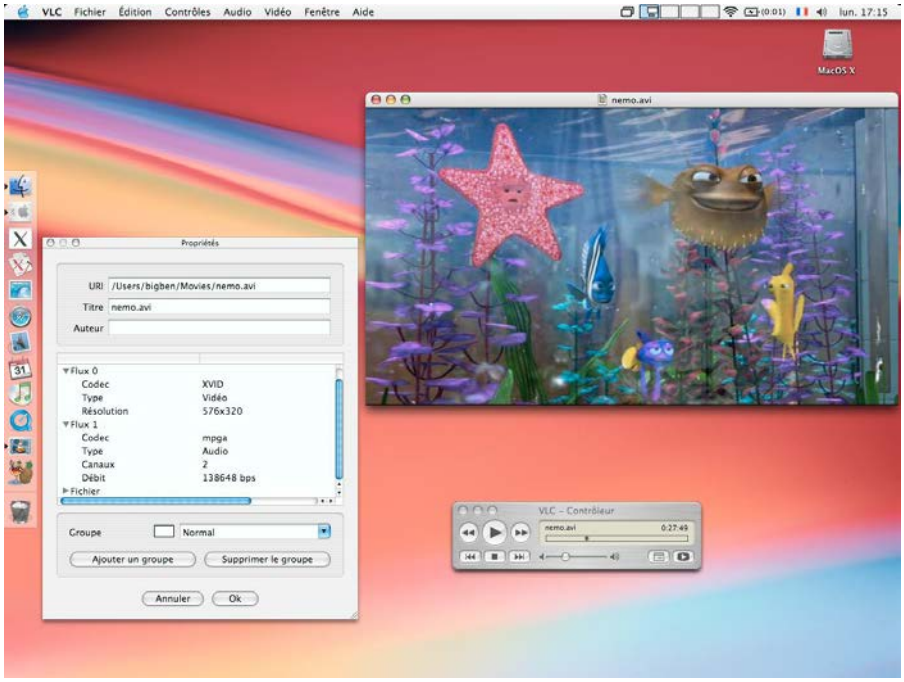
Grafični uporabniški vmesnik (graphical user interface - GUI) je primarno del programa, ki deluje kot povezovalni člen med programom in njegovim uporabnikom. Vsebuje grafične elemente, s katerimi uporabnik upravlja program in grafične elemente, s katerimi program prikazuje stanje (seznam vsebovanih datotek, opozorila …).

Seveda ima lahko tudi operacijski sistem svoj grafični uporabniški vmesnik. Vendar to še ne pomeni nujno, da vsi programi, ki tečejo v tem operacijskem sistemu, tudi vsebujejo GUV. Zaradi poenotenega videza so elementi GUV programov znotraj istega operacijskega sistema praviloma enakega videza in se obnašajo enako, ni pa to nujno.

Na spodnjih dveh slikah (slika 1 in slika 2) vidimo tipična primera GUV pri programih, ki tečeta v operacijskem sistemu Windows 10 in pri OS X.



Slika 1: Grafični uporabniški vmesnik programa VLC v Windows 10 okolju



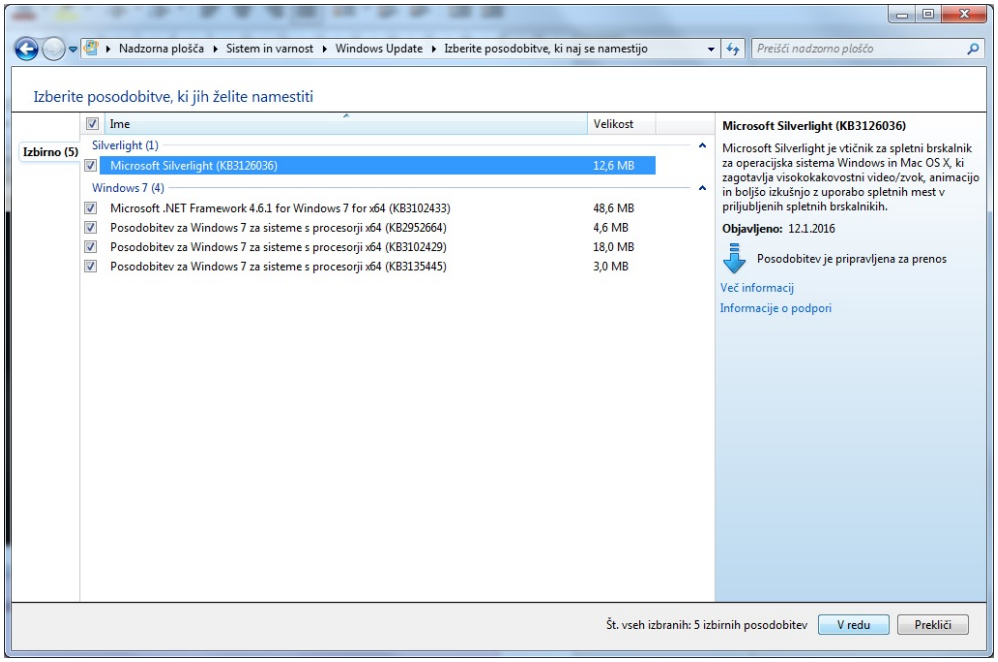
Slika 2: Grafični uporabniški vmesnik programa VLC v Mac X okolju

Za upravljanje programa torej večinoma ne uporabljamo tekstovnih ukazov, ampak uporabljamo interakcijo z grafičnimi elementi. Te grafične elemente po navadi uporabljamo s pomočjo računalniške miške, grafičnega

pisala, v zadnjem času pa tudi preko dotika. Tudi rezultati so pogosto prikazani grafično in ne le v obliki besedila. Na slikah (slika 3 in slika 4) vidimo primerjavo uporabe GUV in tekstovnega vmesnika (ki mu rečemo tudi vmesnik z ukazno vrstico – Command line interface / CLI).

```
login as: uros
uros@192.168.2.5's password:
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-76-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Wed Feb  3 10:59:44 CET 2016

  System load:  0.08               Processes:           116
  Usage of /:   10.5% of 454.27GB  Users logged in:     0
  Memory usage: 19%                IP address for eth0: 192.168.2.5
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

1 package can be updated.
0 updates are security updates.

*** System restart required ***
Last login: Wed Feb  3 10:59:47 2016 from 192.168.2.40
uros@proxyServer:~$ su
Password:
root@proxyServer:/home/uros# apt-get update && apt-get upgrade -y
```

Slika 3: Posodobitev in nadgradnja Linux sistema preko vmesnika z ukazno vrstico



Slika 4: Posodobitev in nadgradnja Windows sistema preko GUV

Ko programiramo svoje aplikacije, običajno predvidimo uporabo grafičnega uporabniškega vmesnika. Kot smo že omenili, so običajno grafični elementi poenoteni v programih znotraj istega operacijskega sistema (pri nekaterih pa obstaja ista shema tudi v več operacijskih sistemih). Načrt grafične postavitve, kot tudi obnašanja GUV, je zelo pomemben del aplikacije, saj s tem vzpostavimo komunikacijo med računalnikom in njegovim uporabnikom. Gradniki dobro načrtovanega vmesnika morajo uporabniku omogočiti zahtevana opravila, s katerimi lahko uporabnik uspešno zaključi delo. Ti gradniki so gumbi, napisi, vnosna polja, izbirna polja ..., ki si jih bomo kasneje podrobneje ogledali in njihovo uporabo razložili na primerih.

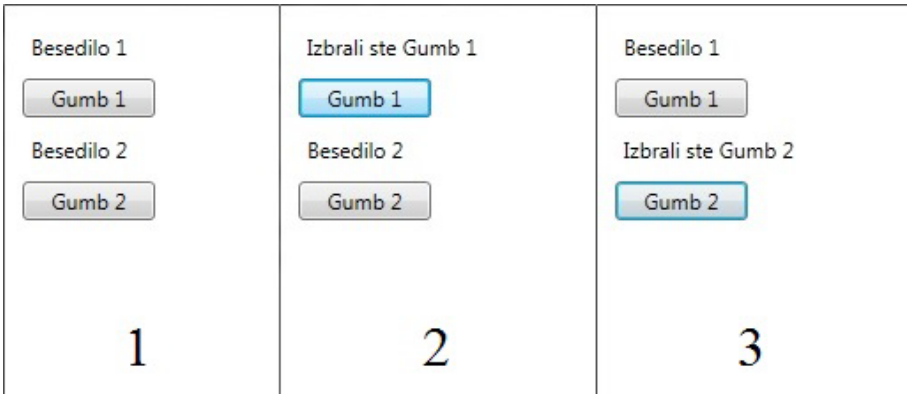
Pri izdelavi GUV moramo biti pozorni tudi na to, da je uporabniku že na prvi pogled vse jasno, kako program deluje. Prav tako se je pri dobrem načrtu treba zavedati omejitev in zmožnosti strojne naprave ter

programske opreme. Tako je npr. pri pametnih telefonih nesmiselno uporabljati miškin kazalec na enak način, kot ga uporabljamo na računalnikih. Tu je veliko bolje, da se uporablja prst oz. več prstov naenkrat (pinch to zoom).

Dogodkovno programiranje

Če program uporabljamo preko grafičnega uporabniškega vmesnika, je delo z njim precej drugačno kot pri uporabi ukazne vrstice. Če uporabljamo zadnjo, se načeloma točno ve, kakšen je potek izvedbe. Zato vemo, da program začnemo izvajati na točno določenem začetku in nato sledimo predvidljivo zaporedje ukazov do konca. Dandanes, ko so večinoma v uporabi GUV, pa je programiranje precej drugačno. Programi z grafičnim uporabniškim vmesnikom čakajo na uporabnikove vhodne podatke, na katere se nato primerno odzovejo. Pri tem vsako uporabnikovo dejanje sproži določen dogodek (`Event`). Te dogodke mora razvijalec pri razvoju predvideti in zapisati postopek, ki se izvede kot odziv na ta dogodek. Ves čas se torej prožijo dogodki in izvajajo določeni postopki, kot odziv na te dogodke. Temu rečemo dogodkovno programiranje. Več informacij o tem lahko najdemo v [1], [2] in [3].

Oglejmo si zgled. Na sliki 5 vidimo aplikacijo, ki vsebuje dva gumba kot grafična gradnika za vnos podatkov in dve oznaki kot gradnika za prikaz teh podatkov. Uporabnik s klikom na posamezni gumb sproži različna dogodka (1 na sliki). Če uporabnik s klikom izbere `Gumb 1`, se besedilo pod njim ne spremeni, vendar se nad njim izpiše, da je bil izbran `Gumb 1` (2 na sliki). V primeru, da uporabnik izbere `Gumb 2`, pa se spremeni spodnje besedilo. Tako se v besedilnem polju izbriše prejšnje besedilo in namesto `Besedilo 2` izpiše `"Izbrali ste Gumb 2."` (3 na sliki).



Slika 5: Sprememba izpisanih besedil v gradniku Label (oznaka) ob kliku na različna gumba

Iz tega razberemo, da za par gradnik/dogodek predvidimo ustrezno navodilo, ki se izvede ob določenem dogodku. Npr. za par `Gumb 1`/klik na ta gumb predvidimo, da se mora spremeniti stanje gradnika `Besedilo 1` (zamenja se napis) in stanje `Gumba 1` (obarva se drugače). Pri paru `Gumb 2`/klik na ta gumb pa se spremeni stanje gradnika `Besedila 2` in `Gumba 2`.

V primeru, da določena akcija sproži več dogodkov, je treba določiti, v kakšnem vrstnem redu se izvajajo. Najprej se sproži en dogodek, ko se ta konča, naslednji in tako do konca.

Integrirano razvojno okolje (IRO)

IRO je programsko orodje, ki je programerjem v pomoč pri razvoju računalniških programov. Tako ga imenujemo, ker je znotraj enega samega okolja združenih več orodij, ki jih potrebujemo za razvoj programov. Najpogosteje so v tem okolju prisotni vsaj urejevalnik kode, prevajalnik in razhroščevalnik.

Večina sodobnih razvojnih okolij temelji na projektnem pristopu k razvoju programov. To pomeni, da je koda obsežnejših programov razdeljena na več delov, saj tako programi pridobijo preglednost. Tako se razvi-

jalec posveča le tistim delom kode, o katerih trenutno premišljuje, ostalih delov pa ne spreminja. IRO podpira tak način dela z različnimi orodji, ki omogočajo enostavno organizacijo delov, hiter preklop med njimi in podobno.

Nekatera IRO okolja omogočajo tudi sprotno preverjanje zapisane kode in njeno sintaktično pravilnost. Pri tem razvijalec že pri pisanju kode prejme opozorilo glede napak in tudi nasvet, kako lahko to napako odpravi.

Poznamo več vrst IRO okolij. Določena so namenjena začetnikom, druga spet bolj izkušenim programerjem. Večina plačljivih okolij je namenjena večjim skupinam izkušenih programerjev, ki izdelujejo programe in jih tržijo. Poleg teh najdemo tudi prosto dostopna IRO okolja, ki so primerna za učenje. Nekatera razvojna okolja, kot je na primer Microsoftov VS 2015 Community, omogočajo tudi programiranje v več različnih programskih jezikih.

Visual Studio 2015 Community

Konec leta 2014 je Microsoft izdal novejšo različico svojega IRO okolja. Ta različica VS Community je podobna prejšnjim različicam VS Professional s to razliko, da je brezplačna. Treba je le pridobiti (prav tako brezplačno) Microsoftov račun. Namenjena je posameznikom in manjšim skupinam, ki se ukvarjajo s programiranjem zaradi učenja, veselja, ali razvijajo manjše aplikacije. S to različico lahko ustvarjamo programe, prirejene za računalnike z operacijskim sistemom Windows, kot tudi programe za druge naprave, ki uporabljajo operacijske sisteme družine Windows, na primer za tablice, mobilne telefone, igralno konzolo Xbox. Poleg te različice sta še dve plačljivi (Professional in Enterprise). Ti dve različici omogočata razvoj poslovnih aplikacij, razvijalcem omogočata vnaprej načrtovano objavo projektov, pregled sprememb v kodi in sledenje zgodovini projekta ... Razlika med plačljivimi in brezplačnimi je tudi v tem, da plačljive podpirajo nekaj tipov projektov, namenjenih poslovnim aplikacijam. Tam so vključeni naprednejši in izboljšani gradniki za povezovanje z bazami podatkov in podobno.

Ena izmed omejitev brezplačne (različice Community) je ta, da na projektu dela največ 5 ljudi, medtem ko pri ostalih dveh teh omejitev ni. Plačljivi različici vsebujeta tudi nabor orodij, s katerimi večje skupine lažje delajo (Team Foundation Server – TFS). TFS vsebuje orodja, s katerimi razvijalci ustvarjajo in urejajo teste, dobijo povratne informacije od uporabnikov aplikacije, dobijo informacije o statusu razvoja aplikacije v obliki grafov in druge. Pri razvoju aplikacije si razvijalci plačljivih različic pomagajo z razporejanjem in prednostnim zaporedjem dela posameznih oddelkov. Pri razvoju, kjer na projektu dela več ljudi, si lahko razvijalci pomagajo tudi z orodjem, ki spremlja spremembe v kodi (CodeLens). To pomeni, da lahko skupina ljudi razvija aplikacijo, ob tem pa so vsi seznanjeni s spremembami, ki so jih delali. Vsi vidijo, kdo je ustvaril spremembo, kdaj jo je ustvaril, ob tem pa lahko ta, ki je spremenil kodo, pusti še opis. Tako razvijalci hitreje najdejo napake in s tem tudi posledično hitreje pridejo do rešitve. Tudi tega orodja ni v Community različici.

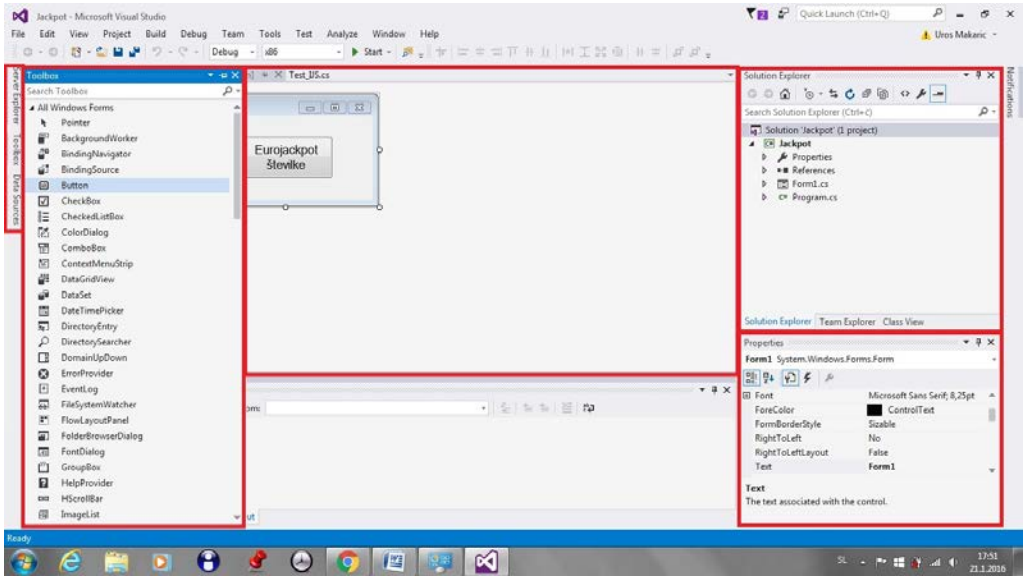
V novem okolju VS lahko razvijamo programe za več operacijskih sistemov. Tako lahko poleg Windows programov in aplikacij sedaj ustvarjamo v VS tudi programe za druge operacijske sisteme (Mac, Linux). Z novim okoljem lahko razvijalci ustvarjajo univerzalne Windows aplikacije, ki jih uporabljamo na računalnikih, tabličnih računalnikih in mobilnih telefonih. Prav tako omogoča novi VS tudi razvoj mobilnih aplikacij za trenutno najbolj razširjene operacijske sisteme za mobilne naprave (Android, iOS …).

Tudi nabor jezikov, ki jih lahko uporabljamo za programiranje, se je v zadnji različici razširil. Poleg jezika C# razvojno okolje podpira tudi jezike Visual Basic, F#, C++, JavaScript, Python in še nekaj drugih.

Na sliki 6 vidimo, da je okolje sestavljeno iz različnih sestavnih delov. Ti so:

- Code Editor (urejevalnik kode)
- Designer (oblikovalec)

- Solution Explorer (raziskovalec rešitve)
- Toolbox (orodjarna)
- Properties (lastnosti)
- Data Sources
- Server Explorer
- Team Explorer
- Class View



Slika 6: Razdelki VS 2015 Community

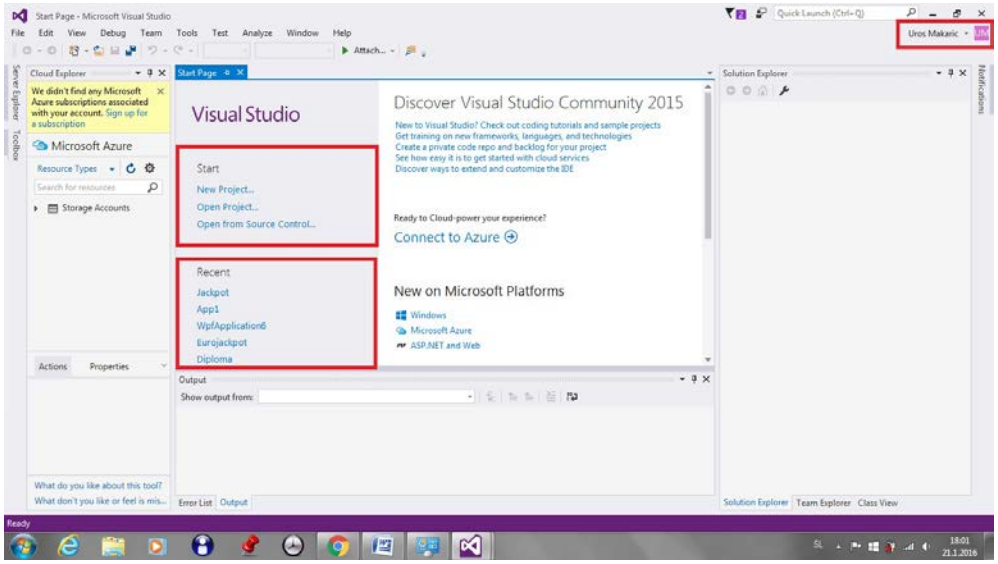
Izdelava projektov

Na sliki 6, desno zgoraj vidimo, da VS ob zagonu poda informacijo o prijavi v VS. Prijavimo se lahko na dva načina. Eden je, da se prijavimo z osebnim računom, ki si ga lahko ustvari vsak. Ta vrsta računa je večinoma primerna za razvijalce, ki jim razvoj aplikacij predstavlja konjiček. Poleg te možnosti pa je še delovni oz. študentski račun. V tega se prijavijo študenti ali razvijalci, ki delajo v skupinah. Večinoma razvijalci v skupinah delajo na večjih projektih. Določene dele teh projektov si nato razdelijo na več manjših, kjer so nekateri odgovorni za grafični del projekta, drugi za programski ...

Čeprav se za delo ni nujno prijaviti, pa prijava prinaša nekaj ugodnosti. Najbolj pomembna je, da lahko razvijalci brezplačno uporabljajo VS (pred prijavo lahko uporabljajo le 30-dnevno testno različico). Poleg tega pa se z vpisom računa v VS okolje samodejno poveže na Azure in Visual Studio Team Services. Podjetja, ki se ukvarjajo z razvojem aplikacij, imajo projekte in njihove datoteke shranjene na strežnikih in ne na vsakem posameznem računalniku. S prijavo v VS lahko razvijalci dostopajo do projektov z različnih računalnikov.

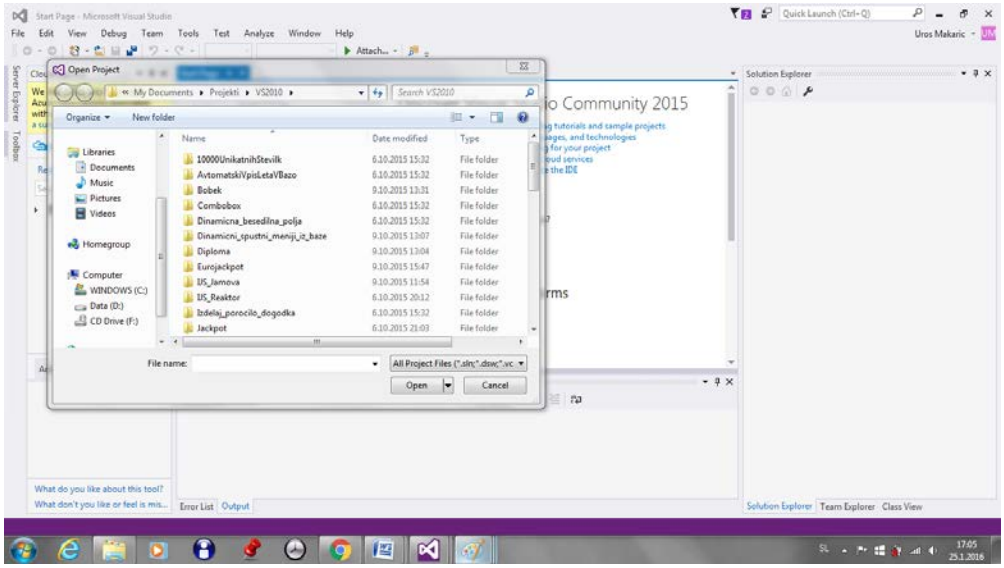
Azure je Microsoftova oblačna storitev za izdelavo, upravljanje in namestitev aplikacij ter storitev preko Microsoftovega omrežja in podatkovnih centrov. S prijavo razvijalci dobijo podporo, učne načrte, določeno brezplačno programsko opremo ... Za izmenjavo kode na projektih, spremljanje opravljenega dela in izdelavo aplikacij pa si razvijalci lahko pomagajo s storitvijo Visual Studio Team Services. Z vpisom v VS se tudi nastavitve VS sinhronizirajo. Ko si razvijalec določene lastnosti VS prilagodi na računalniku, se te lastnosti prenesejo na drugi računalnik, na katerega se razvijalec prijavi (glej npr. [4]).

Na levi strani glavnega okna lahko izberemo, ali bomo ustvarili novi projekt ali pa nadaljevali s kakšnim, ki je že shranjen (slika 7). V primeru, da želimo nadaljevati delo na katerem od predhodnih projektov, ga poiščemo v razdelku `Recent`. Sicer ga poiščemo v razdelku Start, kamor kliknemo na `Open Project ...`



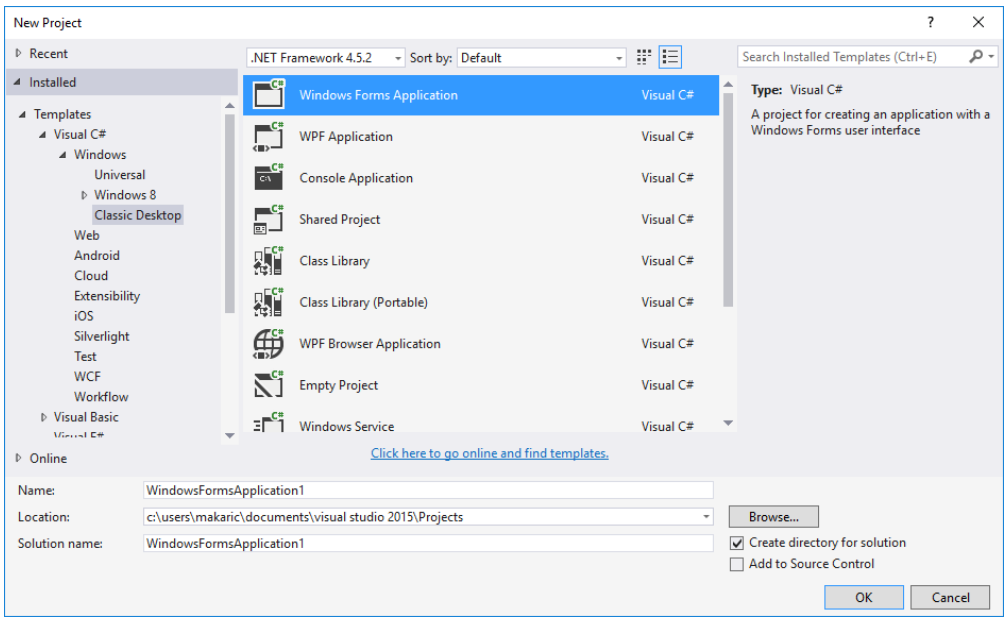
Slika 7: Prijava v Visual Studio Community 2015 in vpis z Microsoftovim računom

Nato se odpre okno s shranjenimi projekti, kjer izberemo tistega, s katerim bomo delali. Znotraj map projektov so datoteke rešitev oz. Solution Files, s katerimi odpremo projekt v VS okolju. Tako lahko nadaljujemo z razvijanjem aplikacije ali pa z njenim testiranjem.



Slika 8: Izbor projekta iz mape, na katerem se nadaljuje delo

Če želimo ustvariti novi projekt, kliknemo ob zagonu VS `New Project` … Ob tem se nam odpre okno, kjer izberemo eno izmed možnosti.



Slika 9: Ustvarjanje novega projekta in izbira vrste projekta (Windows Forms, WPF ...)

Na levi strani okna izberemo programski jezik, v katerem bo potekal razvoj. Ko izberemo npr. C# in kliknemo na Visual C#, se nam odprejo nove možnosti. Tu izbiramo, kakšno aplikacijo bomo ustvarili. Na voljo imamo:

- okenske aplikacije
- mobilne aplikacije
- spletne aplikacije
- aplikacije za okolje android
- aplikacije za okolje iOS ...

Velika večina poslovnih aplikacij, ki jih poznamo, so okenske. Gre za razvijanje programov, ki bodo tekli v okolju Windows in bodo imeli grafični uporabniški vmesnik. Ko izberemo okenske aplikacije, se nam na desni strani zopet ponudi izbor možnosti, izmed katerih izberemo eno.

Okenske aplikacije (Form)

Kot smo že omenili, je velika večina poslovnih aplikacij okenskih, torej takih, ki tečejo v okolju Windows in imajo grafični uporabniški vmesnik. Če smo kot jezik izbrali C#, imamo na voljo tudi pripravo okenskih aplikacij. Med ponujenimi možnostmi nas zanimata prvi dve (Windows Forms in Windows Presentation Foundation), kjer gre za okenski aplikaciji z grafičnim uporabniškim vmesnikom.

Okenska aplikacija je aplikacija, ki vsebuje vsaj en obrazec oz. gradnik tipa Form. Obrazec je ogrodje aplikacije, ki je vidno na zaslonu, in je večinoma pravokotne oblike. Z dodajanjem gradnikov iz orodjarne na obrazec oblikujemo aplikacijo. Pri tem so gradniki lahko taki, ki imajo pomen le za delovanje aplikacije in nimajo svoje grafične podobe (niso vidni v uporabniškem vmesniku) in taki, s pomočjo katerih uporabnik vzpostavi komunikacijo z računalnikom.

V primeru, da ima aplikacija več kot en obrazec, mora eden izmed obrazcev biti glavni oz. `Main Form`. Ta glavni obrazec je povezan z ostalimi obrazci tako, da lahko ostale obrazce odpremo samo preko njega. Zaradi tega je ta glavni obrazec praviloma neprestano odprt. Glavni obrazec običajno vsebuje menije, orodne vrstice ... Ko ga zapremo, se zaprejo vsi ostali obrazci in konča celotna aplikacija.

Okenske aplikacije, ki jih razvijamo v okolju Visual Studio in v jeziku C#, delimo na tiste, ki temeljijo na tehnologiji/pristopu WPF (Windows Presentation Foundation) in na tiste, ki so zasnovane na uporabi Windows Forms. Gradnja aplikacij na osnovi Windows Forms je v uporabi že od začetka leta 2002, pristop, temelječ na WPF, pa se je pojavil konec leta 2006.

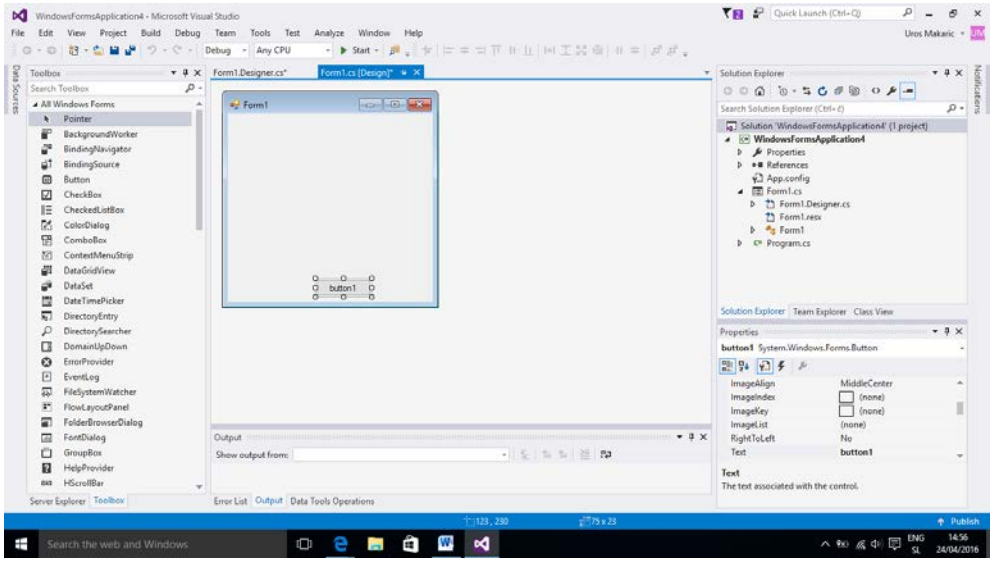
Največja razlika med WPF in Windows Forms je v načinu dela razvijalca aplikacije. Osnova Windows Forms je obrazec, na katerega odlagamo gradnike. Tem gradnikom lastnosti nastavljamo interaktivno v samem VS ali programsko z ukazi, ki jih zapišemo v urejevalniku kode (Code Editor). V zadnjem tudi zapišemo navodila za obnašanje gradnika (kako gradnik reagira na določene dogodke). V vsakem primeru pa so navodila za videz in obnašanje gradnikov zapisana znotraj same programske kode, saj tudi ob uporabi interaktivnega nastavljanja lastnosti, VS ustrezne nastavitve napiše v programsko kodo. WPF aplikacije pa so sestavljene iz objektov, gradnikov in predlog, ki povedo WPF aplikaciji, kam naj postavi te elemente. Ta zapis o postavitvi praviloma urejamo v tekstovnem urejevalniku s pomočjo jezika XAML. Za lažje delo je tekstovni urejevalnik povezan z oblikovalcem (Design), na katerega, tako kot v Windows Forms, odlagamo gradnike. Tako se vsaka sprememba, ki jo naredimo v XAML-u, sočasno odrazi tudi v oblikovalcu. Zaradi uporabe jezika XAML lahko WPF aplikacije naredimo bolj lične in unikatne v primerjavi z Windows Forms aplikacijami. Več o podobnostih, razlikah in možnostih Windows Forms in WPF aplikacij si lahko ogledate v [5].

Ena izmed pomembnejših razlik pri razvoju WPF in Windows Forms aplikacij je prav ta, da je delo pri WPF aplikacijah ločeno na oblikovalski del in razvojni del. Tako pogosto pri razvoju aplikacije sodelujeta dve skupini ljudi. Oblikovalci urejajo in oblikujejo videz grafičnega uporabniškega vmesnika, razvijalci pa delajo na uporabi le-tega. Pri tem oblikovalci lahko delajo v drugem okolju in ne nujno v Visual Studiu. Taka okolja so npr. Blend for Visual Studio, KAXAML in druga ter so prilagojena samemu oblikovanju. Končan projekt oblikovalci nato predajo razvijalcem. Ti grafični uporabniški vmesnik (ustrezne XAML datoteke) prenesejo v Visual Studio. Tam napišejo kodo, kako se vmesnik odziva na različne uporabnikove akcije. S to delitvijo se lahko vsak ukvarja s tistim delom razvoja aplikacije, za katerega je najbolj usposobljen – z oblikovanjem oziroma s kodiranjem (pisanjem programske logike).

Pri WPF aplikacijah razvijalci pripravijo grob osnutek, kjer predvidijo, kateri gradniki bodo potrebni za aplikacijo. Te gradnike nato oblikovalci urejajo. Sočasno, ko se oblikovalci ukvarjajo z grafičnim delom, razvijalci urejajo delovanje aplikacije (programirajo odzive na dogodke, pripravljajo izpise, povežejo določene gradnike …).

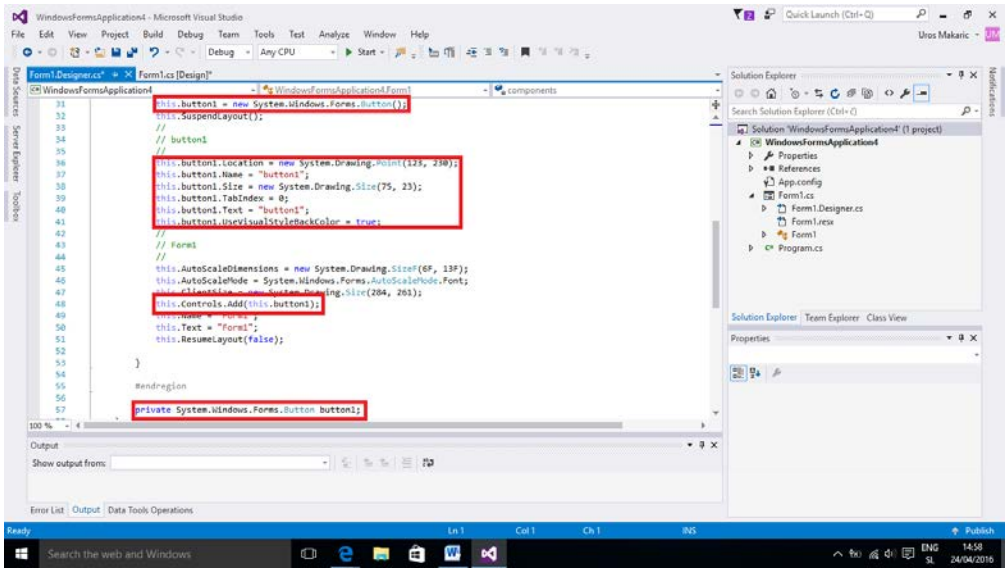
Ker imamo pri WPF aplikacijah več nadzora nad videzom posameznih gradnikov, moramo pri WPF aplikacijah več časa nameniti oblikovanju in urejanju delovanja gradnikov. To pri uporabi Windows Forms ni potrebno. Tam večinoma uporabljamo gradnike v vnaprej pripravljeni obliki.

Pri pripravi preprostih Windows Forms aplikacij večinoma uporabimo način oblikovanja povleci in odloži (slika 10). Pri tem iz orodjarne (`Toolbox`) izberemo gradnik, ki ga želimo vstaviti na okno, in ga povlečemo na izbrano mesto na oknu. Pri tem se v datoteki s končnico `.Designer.cs` samodejno ustvari ustrezen zapis kode (več o tem v razdelku Raziskovalec rešitve).



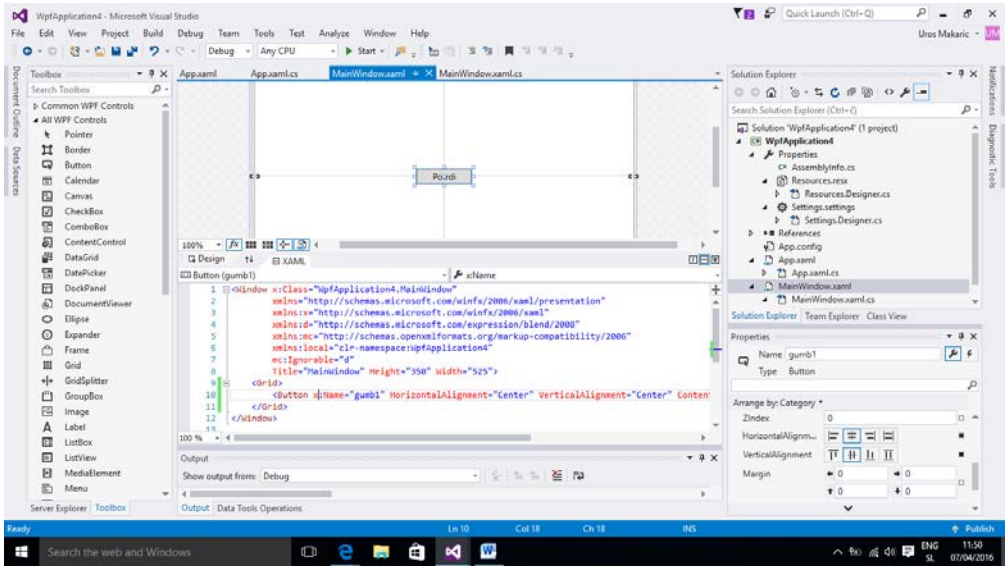
Slika 10: Dodajanje gradnikov iz orodjarne na okno Windows Forms

Na sliki 11 je označen del kode, ki se je ob dodajanju gumba samodejno dodal v datoteki `Form1.Designer.cs`. V kodi, ki je del metode `InitializeComponent`, je najprej zapisan ustrezni ukaz, s katerim se gumb ustvari (prvi označeni del na sliki 11). Nato so na vrsti stavki, s katerimi določimo lastnosti dodanega gumba, kot so velikost, položaj, barva, njegovo ime in morda še kaj. V metodi `InitializeComponent` so torej ukazi, ki ustvarijo gradnike in ukazi za nastavitve lastnosti, ki jih imajo gradniki ob zagonu programa.



Slika 11: Samodejno ustvarjena koda elementov, ki so prisotni na GUV aplikacijah

Kot smo omenili že prej, imajo WPF aplikacije ločen oblikovalski in razvojni del. Pri WPF aplikacijah sta na voljo dva načina razporejanja gradnikov na okna (obrazci). Prva možnost je klasična, ko oblikovalski del urejamo s tehniko povleci in odloži, kot jo uporabljamo pri Windows Forms. Na sliki 12 je v osrednjem delu v XAML urejevalniku zeleno označen zapis, ki ga dobimo, ko odložimo gumb na oknu. Če ga primerjamo z zapisom Windows Forms aplikacije, vidimo, da je tu celoten zapis zapisan v eni vrstici. Zaradi tega je zapis tudi lažje berljiv in urejen. Iz slike vidimo tudi, da lahko tudi pri uporabi XAML spremembe gradnika urejamo preko urejevalnika, ali pa, tako kot Windows Forms, preko lastnosti (desni spodnji kot slike 12).



Slika 12: Dodajanje gradnikov na WPF okno iz orodjarne

Druga možnost pa je oblikovanje videza aplikacije z neposrednim vpisom ukazov jezika XAML. Sprememba zapisa v XAML urejevalniku se odrazi sočasno v oblikovalnem delu.

Denimo da želimo dodati gumb, kot smo ga prej pri opisu sestavljanja obrazca z Windows Forms. V XAML datoteko napišemo znotraj gradnika `Grid`

```
<Button x:Name="gumb1" HorizontalAlignment="Center" VerticalAlignment="Center"
Content="Potrdi" Width="75"/>
```

S tem smo ustvarili gumb, mu določili ime (`gumb1`) in njegovo postavitev. Iz zapisa vidimo, da je gumb znotraj gradnika `Grid` (ukaz se nahaja znotraj značk Grid), in sicer na njegovi sredini. Poleg tega gradniku gumb tudi določimo napis ("Potrdi") in širino.

Takoj, ko ta opis dodamo v XAML urejevalniku, se gumb prikaže na oblikovalnem delu (glej zgornji sredinski del na sliki 12). Do stanja, kot ga prikazuje slika 12, lahko torej pridemo bodisi tako, da gumb povlečemo iz orodjarne in v lastnostih uredimo še postavitev gradnika bodisi da v urejevalniku XAML ustvarimo ustrezni zapis.

Pri WPF aplikacijah se oblikovalski del torej ne zapiše neposredno v obliki kode jezika C# (v datoteko `Designer.cs`), ampak se zapiše v XAML datoteko.

Gradniki, ki sestavljajo WPF aplikacije, so vektorsko načrtovani, medtem ko so gradniki v Windows Forms predstavljeni z bitno sliko. To pomeni, da se pri uporabi WPF vsi elementi aplikacije veliko bolje prilagodijo velikosti in ločljivosti zaslona in pri tem ne pride do izgube kvalitete ter popačenja videza.

Ena od prednosti uporabe WPF pred Windows Forms je tudi ta, da tako oblikovanje gradnika kot njemu pripadajočo programsko kodo, lažje uporabimo pri več aplikacijah. To pride še posebej prav, če moramo razviti več aplikacij, ki se razlikujejo le po uporabniškem vmesniku. Če pri drugi aplikaciji potrebujemo enako oblikovan gradnik, v projekt uvozimo XAML datoteko, kjer je zapisano oblikovanje in dodamo isto ali spremenjeno programsko kodo za ta gradnik. Če pa je treba spremeniti le oblikovanje, naredimo kopijo celotne aplikacije in spremenimo oblikovanje v datoteki XAML. Tako lahko večkrat uporabimo razvojni del programa, kjer so zapisani odzivi posameznih gradnikov. Ta možnost je zelo dobrodošla predvsem za razvijalce, ki enako aplikacijo prodajo več strankam. Spremenijo samo grafični videz v skladu z zahtevami posamezne stranke, logika delovanja programa pa ostane nespremenjena. Ker koda logike aplikacije in koda, ki določa videz aplikacije, nista prepleteni, ampak povsem ločeni, je te spremembe lažje opraviti in z manj napakami.

Iz namizne WPF aplikacije lahko brez večjih težav ustvarimo tudi spletno WPF aplikacijo, ki jo uporabljamo preko spletnega brskalnika. Zaradi vektorsko načrtovanih gradnikov ostane videz spletne aplikacije več ali manj tak kot pri namizni, le prilagojen spletni uporabi, za kar poskrbi samo razvojno okolje VS pri prevajanju. Za pravilno delovanje spletne aplikacije so praviloma potrebne le manjše spremembe določenih delov kode. Več o tem si lahko preberete v [6].

Omeniti moramo še pomembno lastnost WPF. To so predloge, s katerimi lahko ustvarimo znotraj projekta svoj videz in obnašanje gradnikov (gumbov, zapisov, spustnih polj …). Tako lahko namesto klasičnega videza gradnika aplikacije ustvarimo svojega, ki ga lahko naknadno še spreminjamo.

S predlogami si lahko pomagamo, ko želimo nekemu gradniku spremeniti videz, obnašanje, ali pa želimo ustvariti povsem novi gradnik, ki ni na voljo v orodjarni. Če imamo v aplikaciji več podobnih gradnikov (npr. gumbov) in jim želimo spremeniti npr. barvo ozadja ali obliko, bi bilo urejanje sprememb na vsakem gumbu posebej dolgotrajno. Poleg tega pa bi lahko na kakšen gumb v aplikaciji tudi pozabili in potem v aplikaciji ne bi imeli enotnega videza. Prav tako bi morali vsakemu na novo dodanemu gumbu spet nastavljati te lastnosti. Če pa ustvarimo predlogo gumba in vse gumbe potem ustvarimo na osnovi te predloge, se vse lastnosti gumbov v aplikaciji spremenijo takoj, ko spremenimo predlogo.

Predlogo projekta ustvarimo v VS tako, da ob izbiri vrste projekta izberemo `WPF User Control Library` in ga poimenujemo npr. `WPFPredloga`. Nato z desnim miškinim klikom na `WPFPredloga` iz seznama izberemo `Add/Resource Dictionary` in vpišemo ime, npr. `Tema`. V tej datoteki nato določimo celotno temo aplikacije (videz gradnikov, animacija gradnikov, barva gradnikov …).

Kot primer si bomo pogledali, kako ustvarimo predlogo za gumb. V datoteki `Tema.xaml` z naslednjo kodo spremenimo videz gradnika Button:

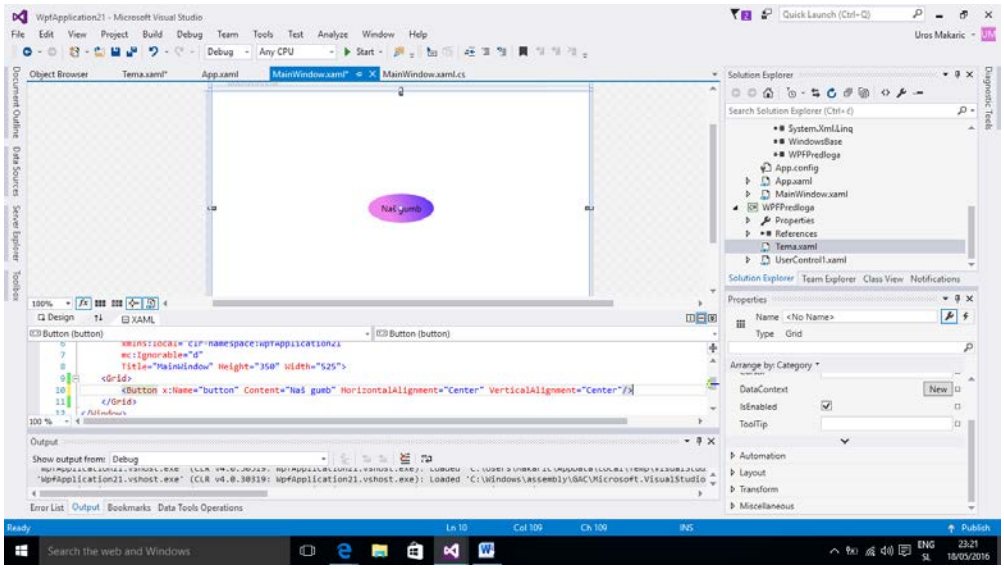
```
<Style TargetType="Button">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType = "Button">
                <Grid>
                    <Ellipse x:Name = "ButtonEllipse" Height="40" Width="90">
                        <Ellipse.Fill>
                            <LinearGradientBrush StartPoint = "1.5,0.5"
                                  EndPoint = "0.5,1.5">
                                <GradientStop Offset = "0" Color = "Blue" />
                                <GradientStop Offset = "1" Color = "Violet" />
                            </LinearGradientBrush>
                        </Ellipse.Fill>
                    </Ellipse>
                    <ContentPresenter Content = "{TemplateBinding Content}"
                         HorizontalAlignment = "Center" VerticalAlignment = "Center" />
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Z značko `Style` napovemo nov stil oz. ureditev gradnika, kjer povemo, kateri gradnik bomo urejali (v našem primeru gumb). Z naslednjo značko povemo, da naš gumb povozi splošne nastavitve gumba in nastavitve, navedene s tem opisom, uporabi za privzete za gradnik Button. Naslednja značka definira obnašanje in grafični del gradnika. V našem primeru oblikujemo gumb kot elipso z določeno višino in širino. Poleg tega nastavimo tudi barvo gradnika. V predstavljenem primeru smo uporabili prelivanje dveh barv znotraj gumba. V znački `ContentPresenter` zapišemo, da bomo napis na gumbu določili kasneje. Dodamo še zapis, ki postavi besedilo gumba na sredino.

Da bomo to predlogo lahko uporabili v svojem programu, jo moramo vključiti v rešitev aplikacije. To naredimo tako, da z desnim miškinim klikom na `Solution 'ime_aplikacije'` iz spustnega polja izberemo `Add` in nato še `Existing Project`. V oknu, ki se nam odpre, poiščemo in izberemo projekt predloge `WPFPredloga.csproj` in kliknemo na `Open`. Poiščemo in odpremo datoteko `App.xaml`. Znotraj datoteke poiščemo značko `Application.Resources` in vanjo vpišemo spodnjo kodo. Z njo povemo, kje so podatki o predlogi.

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="/WPFPredloga;component;/Tema.xaml"/>
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

Da bo zgornja koda delovala, je treba v projekt vključiti tudi referenco, preko katere bo aplikacija prepoznala predlogo `WPFPredloga`. To naredimo tako, da z desno miško kliknemo na `References` in izberemo `Add Reference` … V oknu, ki se nam odpre, obkljukamo predlogo `WPFPredloga` in kliknemo `OK`. Sedaj je v orodjarni že naš spremenjeni gumb. Tega kot prej lahko prenesemo na okno (slika 13).



Slika 13: Dodajanje našega gumba iz predloge na okno

Podobno si lahko v WPF aplikacijah pomagamo s stili. Ti so precej podobni predlogam, le nekoliko manj nastavitev lahko opravimo. Več o predlogah in stilih si lahko preberete v [7].

Tudi tu ustvarimo predloge stilov. Te so lahko vezane na posamezen obrazec (okno), vendar nam časovno najmanj časa vzame urejanje, če predlogo stila projekta ustvarimo v datoteki `App.xaml`. V tej datoteki lahko pripravimo predlogo stila za celotno aplikacijo.

Kot zgled opravimo enako spremembo gumba, kot smo ga opravili prej s pomočjo predloge. Torej želimo, da gumbi niso pravokotne oblike, ampak da so zaobljeni.

V datoteko `App.xaml` znotraj značke `Application.Resources` dodamo:

```
<ControlTemplate x:Key = "PredlogaStilaGumba" TargetType = "Button">
    <Grid>
        <Ellipse Height = "40" Width = "90" >
            <Ellipse.Fill>
                <LinearGradientBrush StartPoint = "0,0.2" EndPoint = "0.2,1.4">
                    <GradientStop Offset = "0" Color = "Blue" />
                    <GradientStop Offset = "1" Color = "Violet" />
                </LinearGradientBrush>
            </Ellipse.Fill>
```

```
        </Ellipse>
        <ContentPresenter Content = "{TemplateBinding Content}"
                HorizontalAlignment = "Center" VerticalAlignment = "Center" />
    </Grid>
</ControlTemplate>
```

Gumbu smo s tem določili obliko, velikost in barvo. Tudi tu na enak način, kot smo to storili pri predlogi, povemo, da bomo napis na gumbu določili kasneje in bo različen za vsak posamezni gumb.

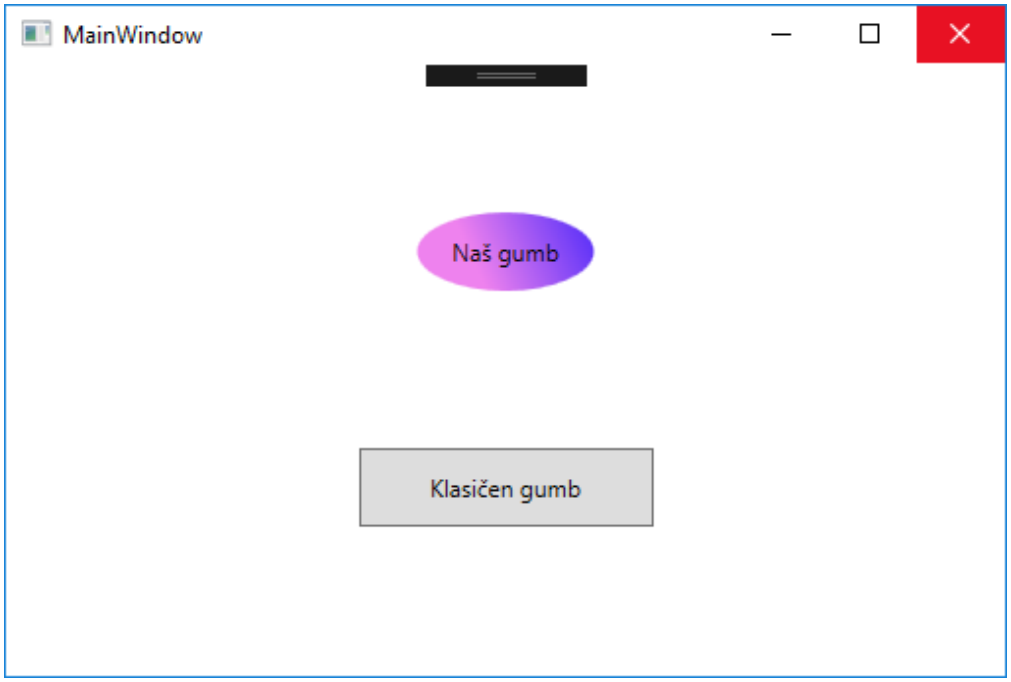
V orodjarni ni sprememb. Ko gumb povlečemo na okno, je videti enako kot prej. V XAML datoteki moramo

```
<Button>Naš gumb</Button>
```

spremeniti v

```
<Button Template = "{StaticResource PredlogaStilaGumba}">Naš gumb</Button>
```

S tem povemo, da sta oblika in videz tega gumba temelj iz definicije, ki je zapisana v `PredlogaStilaGumba`. Na sliki vidimo razliko med klasičnim gumbom, ki nam ga ponuja Visual Studio, in gumbom, urejenim s stilsko predlogo.



Slika 14: Primerjava klasičnega gumba in gumba, ustvarjenega s stilsko predlogo

Slabost tega načina je, da kadar želimo to stilsko predlogo uporabiti v kakšnem drugem projektu, moramo tisti del datoteke `App.xaml`, kjer je opis predloge, kopirati in jo prilepiti v datoteko `App.xaml` drugega projekta. Ker je v datoteki `App.xaml` zapisana tudi druga informacija (npr. katero je glavno okno aplikacije), moramo paziti, da pri tem kopiranju ne naredimo kake neželene spremembe. Več informacij o predlogah in stilskih predlogah, njihovem ustvarjanju in urejanju, najdemo v [8], [9], [10] in [11].

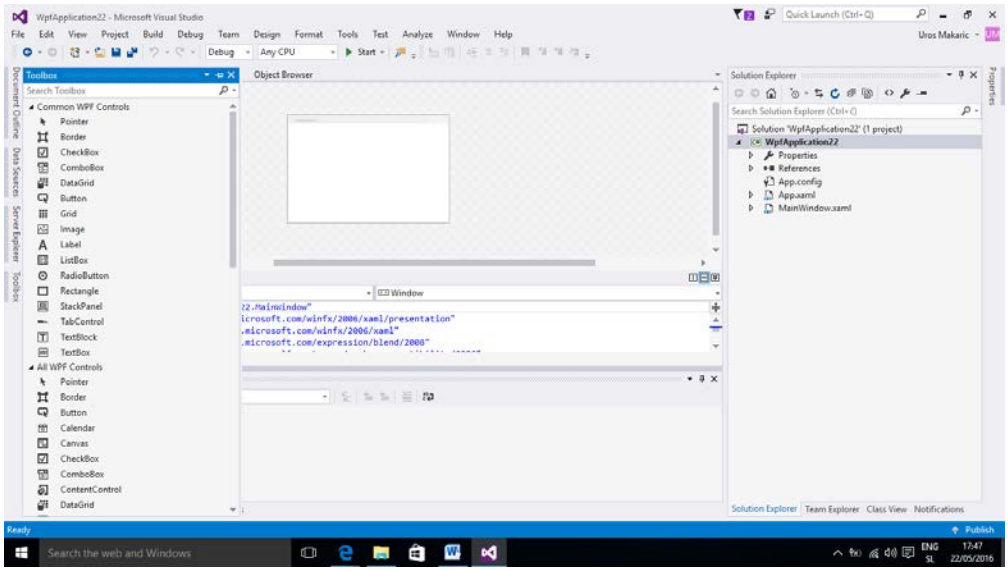
Če smo zadovoljni z videzom gradnikov, ki jih ponuja Windows Forms, je verjetno lažje uporabljati ta način dela. A to velja le, če bomo več ali manj uporabljali privzete oblikovalske lastnosti (recimo tip na gradniku uporabljene pisave). Če pa želimo, da ima naša aplikacija svoj prepoznavni videz, je bolje uporabiti WPF.

Poleg razlik v oblikovanju je ogrodje WPF prineslo še druge novosti. Ker je ogrodje WPF novejše, je bolj prilagojeno novejšim standardom. Zaradi tega v aplikacijo lažje vključimo animacije, 3-D vsebine in video vsebine. Pomemben dodatek je tudi podatkovna vezava, ki si jo bomo ogledali ob spoznavanju gradnika `DataGrid`.

Prednost Windows Forms je v glavnem v tem, da je ta način razvoja aplikacij dlje časa v uporabi. Zato je podrobno preizkušen s strani razvijalcev. Poleg tega je na voljo veliko število dodatnih gradnikov, s katerimi si poenostavimo delo. Velika prednost Windows Forms je tudi obsežnejša dokumentacija, ki je dostopna na internetu. Več o podobnostih in razlikah med WPF in Windows Forms aplikacijami najdemo na primer v [12].

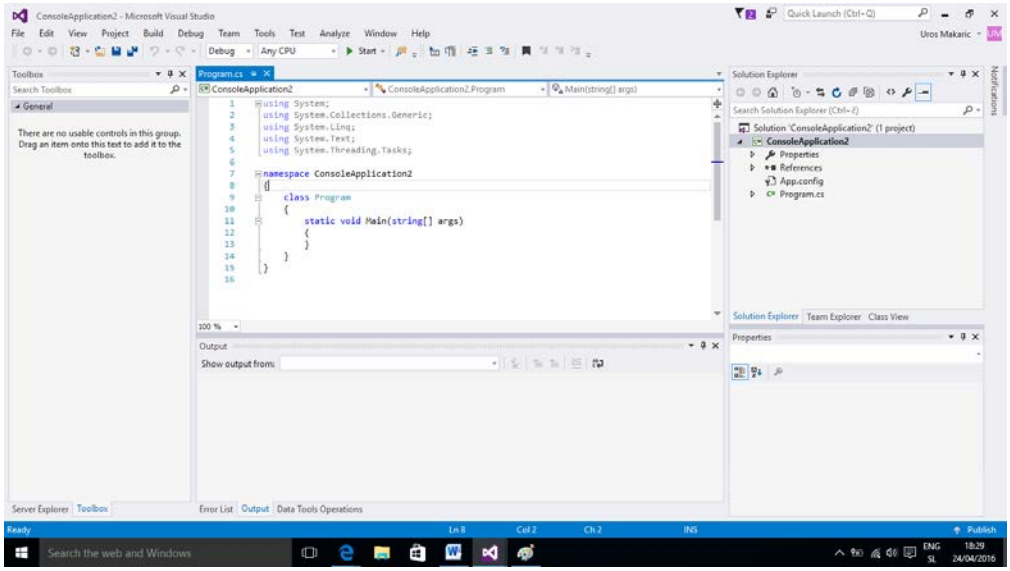
Urejevalnik kode (Code Editor)

Najbolj pomemben del okolja je urejevalnik kode, ki zavzema osrednji del. A če smo se odločili, da uporabljamo Windows Forms ali WPF, se po izboru tipa aplikacije okolje prikaže tako, kot vidimo na sliki 15. Tu urejevalnik kode sploh ni viden, saj se predvideva, da bomo najprej ustvarili uporabniški vmesnik. Na levi strani imamo orodjarno, kjer imamo vrsto gradnikov, ki jih lahko uporabimo pri gradnji uporabniškega vmesnika.



Slika 15: WPF aplikacija, kjer v osrednjem delu ni urejevalnik kode

Če pa na začetku izberemo konzolno aplikacijo (torej tako brez GUV), se bo okolje zagnalo tako, da bo urejevalnik zavzemal osrednje okno (alika 16). Na sliki tudi vidimo, da takrat v orodjarni (Toolbox) ni nobenega gradnika.



Slika 16: Osrednji del okna pri konzolnih aplikacijah zavzema urejevalnik kode

Pri razvoju Windows Forms in WPF aplikacij se med urejevalnikom kode in grafičnim oblikovalcem posameznega obrazca premikamo s tipkama `Shift` in `F7`. Ko smo v urejevalniku kode, se s kombinacijo tipk `Shift + F7` odpre grafični oblikovalec določenega obrazca. Iz grafičnega oblikovalca pa lahko nazaj v urejevalnik kode preklopimo s tipko `F7`. Preklop lahko izvedemo tudi preko menijske vrstice, kjer izberemo `View`. Tu nato izbiramo med urejevalnikom kode (`Code`) ali oblikovalcem (`Designer`). Še tretji način preklopa pa je uporaba desnega gumba v oblikovalcu. S tem odpremo okno, kjer izbiramo med urejevalnikom (`Code`) in oblikovalcem (`Designer`).

Ne glede na to, kakšen tip aplikacije izberemo, okolje samodejno napiše nekaj kode. Kaj in katero je odvisno od tipa aplikacije.

Pri konzolni aplikaciji dobimo ogrodje, kot ga prikazuje vsebina urejevalnika kode na osrednjem delu na sliki 16. Visual Studio je samodejno ustvari datoteko `Program.cs` in v njej omenjeno kodo. V kodi je definiran razred `Program`, ki vsebuje metodo, imenovano `Main`. Nad tem zapisom pa vidimo stavke, s katerimi v projekt vključimo določene imenske prostore (npr. `using System`).

Če smo kot tip aplikacije izbrali Windows Forms ali WPF, je dobljeno ogrodje drugačno. Prav tako ne dobimo le ene datoteke s kodo, ampak več. Podrobneje bomo dobljena ogrodja in vsebino datotek predstavili v razdelku raziskovalec rešitve (Solution Explorer).

V urejevalniku kode pišemo in urejamo kodo. V njem ustvarimo razrede, funkcije, odzive na dogodke, zapišemo potek delovanja aplikacije, vire, iz katerih aplikacija pridobi podatke … Urejevalnik ponuja določene postopke in pripomočke, s katerimi poenostavimo in pohitrimo pisanje kode programov. Oglejmo si nekatere od teh postopkov. Omenili bomo samodejno barvanje, zamikanje, skrivanje delov kode, zaznamke in sprotno prevajanje.

Pri pisanju kode se rezervirane besede samodejno barvno označijo. Na sliki 17 vidimo, kako so modro obarvane rezervirane besede `string`, `switch`, `case`, `true` in `break`. Na ta način se lažje izognemo temu, da bi kot ime neke spremenljivke po pomoti uporabili rezervirano besedo. Prav tako so z enotno barvo označeni tudi drugi sestavni deli. Tako npr. na sliki 17 vidimo tudi, da so vsi nizi napisani z rdečo, komentarji pa s svetlo zeleno …

V C# zamikanje kode ni obvezno (kot je npr. v Pythonu). Vendar z zamikanjem omogočimo boljši pregled nad delovanjem kode. Pri tem nam pomaga urejevalnik, saj se pri pisanju koda samodejno zamika. Ob uporabi zavitega predklepaja ({), urejevalnik položaj za pisanje samodejno zamakne in doda še zaklepaj.

```
string vzrok = dr["vzrok"].ToString(); //iz baze dobimo vzrok
switch (vzrok) // preverimo kateri izbor je izbran
{
    case "požar":
        {
            radioButton1.Checked = true; // če je v podatkovni bazi vzrok 'požar', označimo na obrazcu požar in končar
            break;
        }
    case "vlom":
        {
            radioButton2.Checked = true;
            break;
        }
    case "vklop sistema":
        {
            radioButton3.Checked = true;
            break;
        }
    case "žival":
        {
            radioButton4.Checked = true;
            break;
        }
```

Slika 17: Primer označevanja rezerviranih besed, nizov in samodejnega zamika

Urejevalnik kode v Visual Studiu omogoča tudi prikazovanje in skrivanje delov kode. Samodejno je to omogočeno za prikaz vsebine imenskih prostorov, razredov in funkcij, v razdelku Označevanje in skrivanje območij pa si bomo ogledali, kako lahko prikazujemo/skrivamo poljubne dele kode.

Na sliki 18 vidimo, da so ob levi strani urejevalnika za številkami vrstic prikazani znaki –. Ti označujejo bloke kode. Z levim miškinim klikom lahko del kode, ki spada v ta blok, skrijemo. Ob tem se na levi strani urejevalnika namesto znaka – pojavi +. Na ta način pridobimo pri preglednosti obsežnejših datotek s kodo, saj lahko ustrezne dele kode, ki nas v tistem trenutku ne zanimajo, skrčimo. Razširjen ohranimo le tisti del kode, na katerem trenutno delamo. Za ostale dele pa vidimo le glave določenih funkcij ali metod, lahko pa skrijemo kar vse podrobnosti določenega razreda. Kadar želimo na hitro videti vsebino skritega dela, se le z miško zapeljemo na konec metode, funkcije …, kjer je pravokotnik in v njem tri pike. Takrat se v okencu izpiše zakrita koda. Kot vidimo na sliki 18, zagledamo pravokotnik, v katerem imamo zapisano kodo ob dogodku `button_Click`.

```
17  {
18      /// <summary>
19      /// Interaction logic for MainWindow.xaml
20      /// </summary>
21      public partial class MainWindow : Window
22      {
23          public MainWindow()
24          {
25              InitializeComponent();
26          }
27
28          //dogodek ob kliku na gumb
29          private void button_Click(object sender, RoutedEventArgs e)...
35
36          //funkcija sešteje števili zapisani v dveh besedilnih poljih
37          private string Sestej(string a, string b)
38          {
39              int rezultat = Int32.Parse(a) + Int32.Parse(b);
40              return rezultat.ToString();
41          }
42      }
```

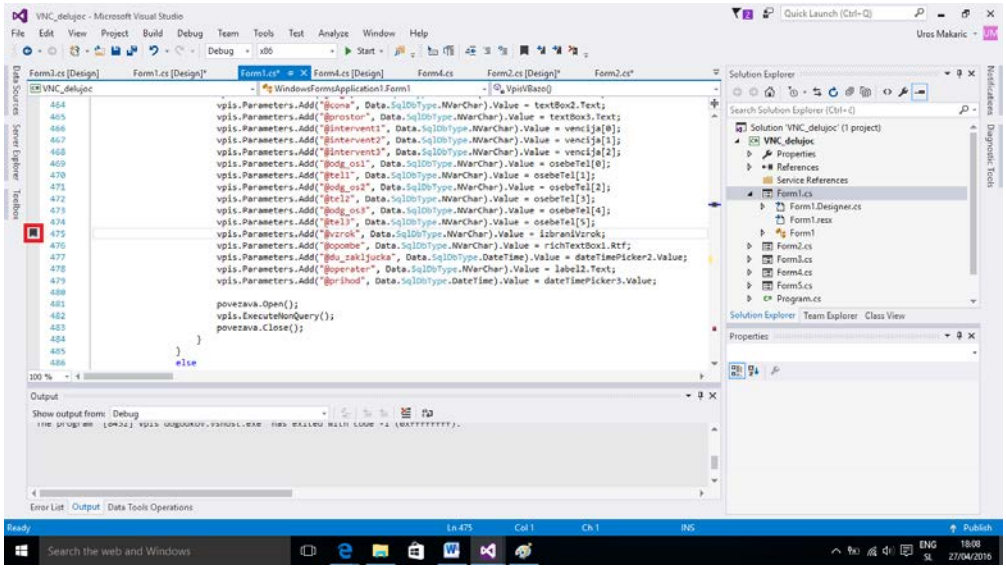
```
private void button_Click(object sender, RoutedEventArgs e)
{
    rezultat1.Content = Sestej(textBox1.Text, textBox11.Text);
    rezultat2.Content = Sestej(textBox2.Text, textBox21.Text);
    rezultat3.Content = Sestej(textBox3.Text, textBox31.Text);
}
```

Slika 18: Primer skritega dela vsebine kode

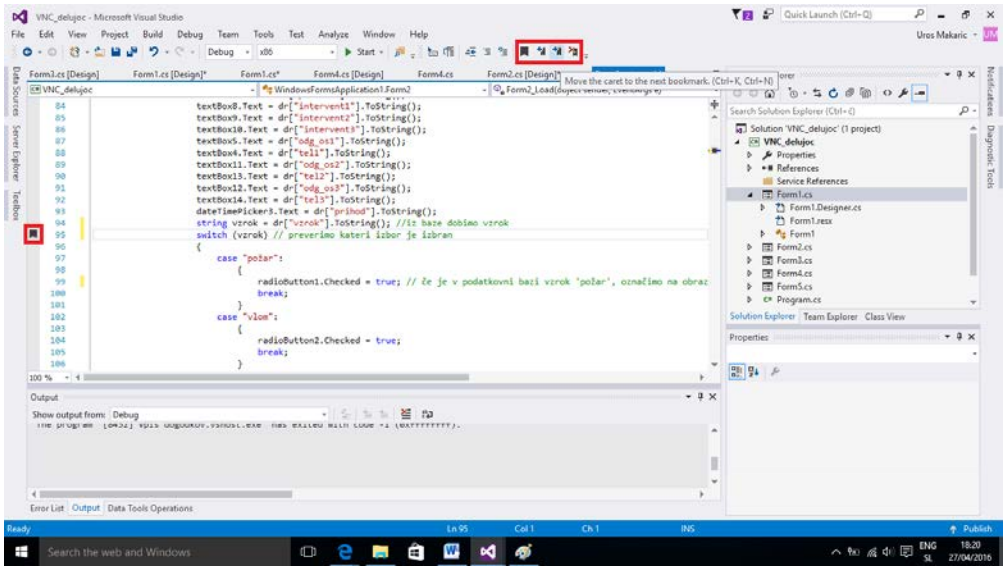
Pri obsežnejših programih se moramo občasno pri delu sprehajati iz enega dela kode v drugega. Pri tej navigaciji med posameznimi deli kode si pomagamo z zaznamki (`Bookmarks`). Na sliki 19 vidimo na levi strani označen zaznamek, ki ga postavimo v urejevalniku kode. Tako se lahko hitro premaknemo na točno določen del kode. Zaznamki niso omejeni ne na datoteko ne na projekt. Tako imamo lahko v raziskovalcu rešitev odprtih več projektov ali obrazcev in na vsakem od njih v različnih delih označene zaznamke.

Zaznamke lahko pregledujemo tudi preko okna zaznamkov. Do njega pridemo tako, da v VS kliknemo v menijski vrstici na `View`, kjer poiščemo `Bookmark Window`. Znotraj tega okna lahko urejamo postavitev zaznamkov in njihovo sosledje.

Tako se v primeru, da dodamo prvi zaznamek v datoteki `Form1.cs`, drugega in tretjega v datoteki `Form2.cs` in četrtega v datoteki `Form3.cs`, tudi sprehajamo po tem vrstnem redu med zaznamki.



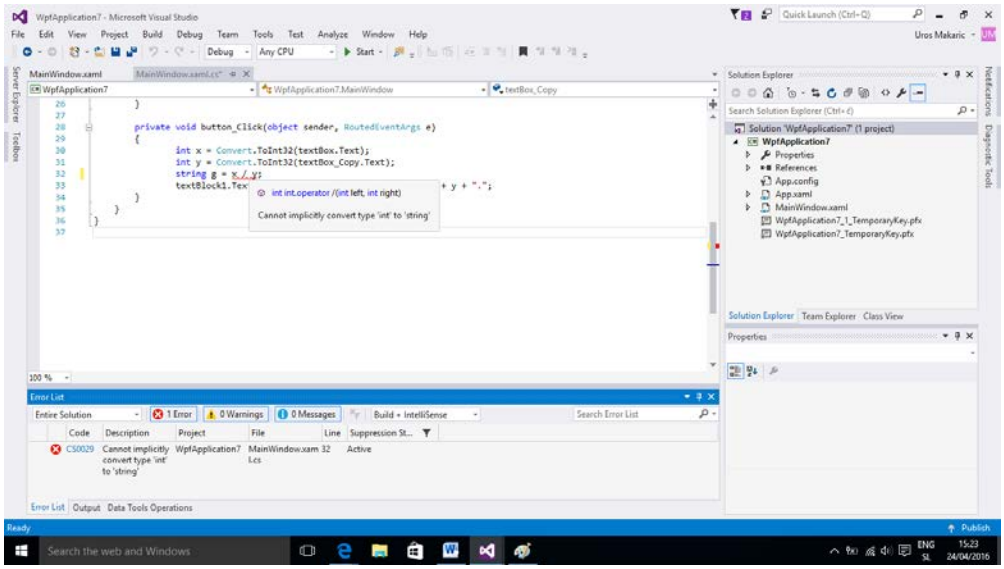
Slika 19: Zaznamek v vrstici 475 kode obrazca Form1



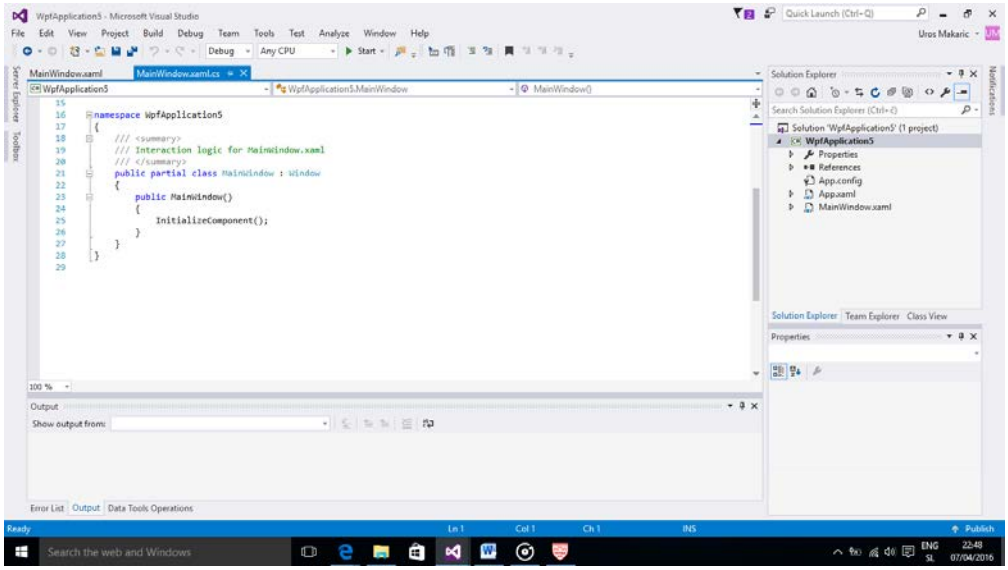
Slika 20: Dodan zaznamek v 95. vrstici obrazca Form2

Nove zaznamke postavljamo s klikom na levi gumb skupine, označen na sliki 20 z rdečim pravokotnikom. Poleg njega so še trije gumbi. Prvi nas premakne na mesto naslednjega zaznamka, njegov sosed pa na prejšnji zaznamek. S klikom na zadnji gumb izbrišemo zaznamek v vrstici, v kateri smo.

Naslednja pomoč, ki nam jo nudi razvojno okolje, je skrb, da je zapisana koda ves čas sintaktično pravilna. Za to, da je pravilna že med samim nastajanjem, skrbi sočasno prevajanje kode. Prevajalnik med tem, ko pišemo v urejevalnik, preverja, ali je zapisana sintaksa pravilna. V primeru sumljive kode nas na to takoj opozori tako, da ta del kode podčrta z rdečo vijugo. Če se z miško premaknemo nad to vijugo, se nam izpiše obvestilo, za kakšno napako gre (slika 21). Seveda je razlog za napako lahko tudi drugačen in je nastal že prej.



Slika 21: Izpis obvestila o napaki, ko se z miško premaknemo nad napako



Slika 22: Urejevalnik kode v razvojnem okolju Visual Studio

Poleg opisanega imamo v urejevalniku kode na voljo še več prijemov in pripomočkov. Med njimi omenimo

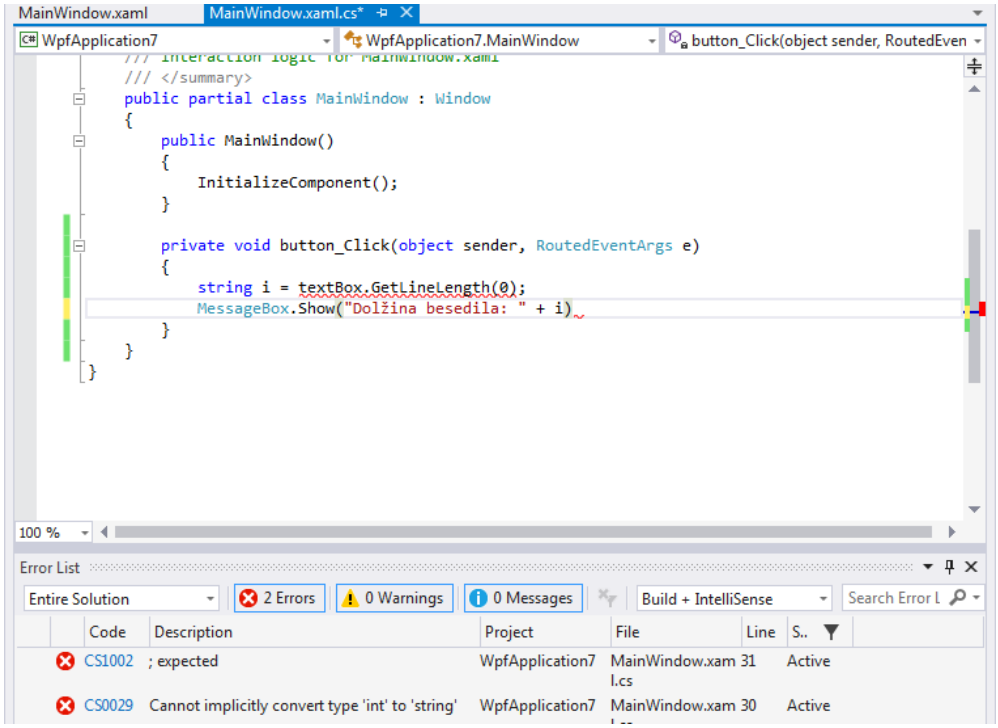
- Refactoring (predelovanje kode)
- Code Snippets (uporaba koščkov kode)
- Code Colorization (obarvanje kode glede na pomen)
- IntelliSense (inteligentno prepoznavanje kode)

V nadaljevanju si bomo podrobneje pogledali pripomoček IntelliSense. Podrobnejše informacije o urejevalniku kode pa dobite, na primer v [13].

IntelliSense

Eno izmed orodij v VS, ki bistveno pripomore k enostavnosti in hitrosti dela, je IntelliSense. S tem imenom označujemo številne funkcije urejevalnika kode, kot so sprotno preverjanje pravilnosti sintakse, prikazovanje informacij o kodi in podobno. Med funkcijami orodja IntelliSense najpogosteje srečamo ([14]) izpis imen vseh članov in metod razredov (`List Members`), prikaz informacij o parametrih metod (`Parameter Info`), dostop do hitre pomoči (`Quick Info`) in samodokončanje (`Complete Word`). S temi funkcijami je pisanje kode preprostejše in hitrejše. Za več informacij o IntelliSensu v okolju Visual Studio si oglejte [14], [15] in [16].

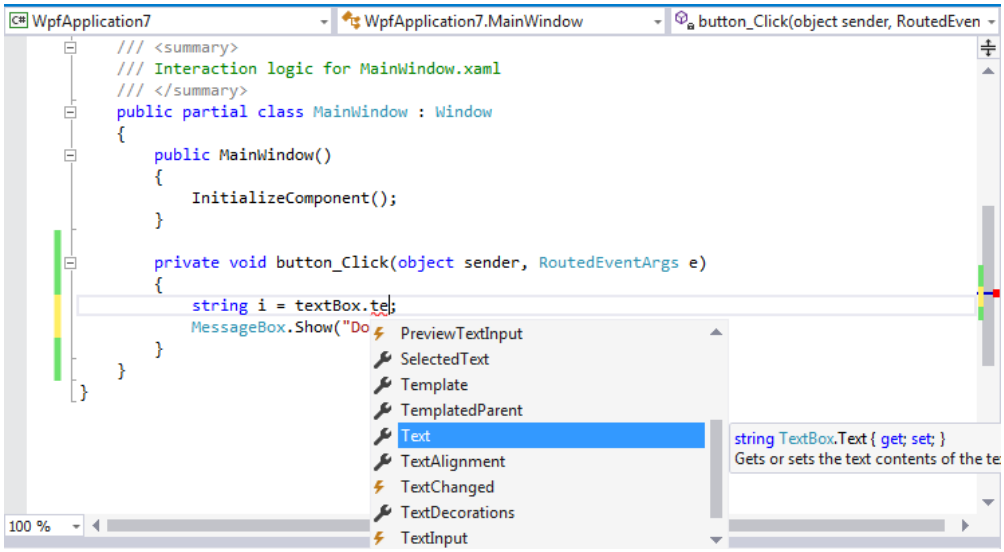
Tudi sprotno preverjanje pravilnosti kode, o katerem smo že govorili, pogosto štejemo kot eno od funkcij orodja IntelliSense. Pomaga nam pri pravilnem zapisu sintakse, saj neprestano preverja sintaktično pravilnost zapisane kode. Kot smo omenili, nas v primeru napake v kodi opozori, kje je prišlo do nepravilnosti in predlaga, kako lahko to napako odpravimo ([15]). Na sliki 23 vidimo, kako smo v spodnjem delu urejevalnika kode dobili obvestilo o sintaktični napaki, njen opis, in v sami kodi z rdečo vijugo podčrtano mesto opažene napake. Seveda moramo vedeti, da ni nujno, da je napaka res taka, kot je sistem predvidel. Lahko je napaka storjena že prej, a od tu dalje prevajalnik le ni znal ustrezno nadaljevati s prevajanjem. V predstavljenem primeru je bilo okolje uspešno, saj pravilno predvideva zaključek stavka in nas opozori, da manjka podpičje. Poleg tega nas opozori tudi na to, da ne more implicitno pretvoriti celega števila v niz. Označeno je tudi, da sta napaki v 30. in 31. vrstici kode.



Slika 23: Prikaz napak v kodi

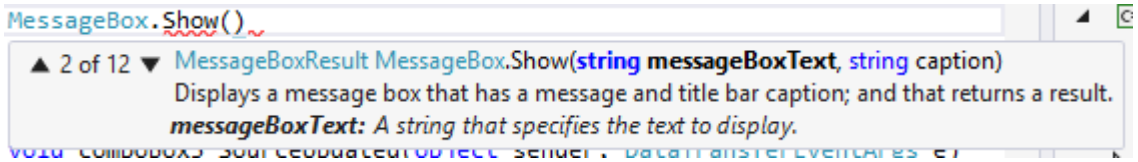
Zelo uporabno je samodokončanje. Ko začnemo tipkati, nam okolje samo ponudi tiste sestavne dele kode, ki se začno z natipkanim delom. Pri tem okolje upošteva kontekst, torej ponudi le tista imena spremenljivk, funkcij in podobno, ki v danem trenutku pridejo v poštev. Na sliki 24vidimo, kako lahko razvijalec že z nekaj zapisanimi črkami prikaže lastnosti in metode določenega razreda. Seznam članov (lastnosti in metod) se pojavi potem, ko po zapisu imena spremenljivke, ki predstavlja objekt nekega razreda, natipkamo piko. Takrat se prikaže spustno polje, ki prikaže seznam lastnosti in metod tega razreda. Ko razvijalec doda še kakšno črko, se začne ta seznam manjšati in prikazuje samo člane, ki se začno z zapisanim nizom znakov. Po ponujenem seznamu se lahko premikamo s tipkama navzgor ali navzdol. Ob tem na desni strani seznama

vidimo kratek opis metode oziroma lastnosti. Ko označimo metodo s seznama, s pritiskom na tipko presledek ali na tipko enter vnesemo preostanek imena metode oziroma lastnosti. Zaradi te možnosti se pri zapisu kode število napak bistveno zmanjša, saj v seznamu ni prisotnih metod, funkcij, spremenljivk, ki jih v tem trenutku ne moremo uporabiti. Poleg tega pa tudi hitreje napišemo kodo, saj lahko s samo nekaj zapisanimi znaki vnesemo daljša imena. Uporaba samodokončanja tudi omogoča, da lažje uporabljamo daljša imena spremenljivk in metod ter s tem prispevamo k razumljivosti kode.



Slika 24: Izpis vseh imen članov in metod razredov, ki se začno z nizom te

V veliko pomoč pri razvoju je tudi informacija o parametrih metode (parameter Info). Ta funkcija nam ob imenu metode prikaže informacije o številu, imenu in tipu parametrov, ki so potrebni pri metodi. V oknu se pokažejo vse možne oblike klica te metode. Takoj po vnosu imena metode in predklepaja se pojavijo možnosti izbora parametrov, ki jih lahko uporabimo pri klicu.



Slika 25: Primer prikaza informacije o parametru metode v okolju Visual Studio

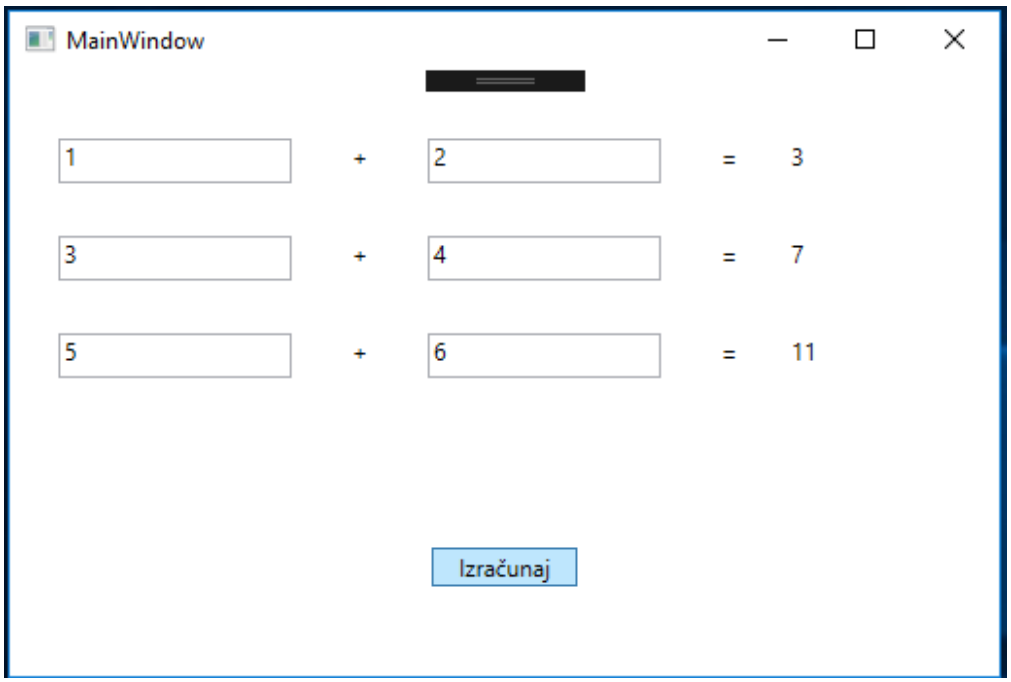
Na sliki 25 vidimo možnosti, ki jih lahko uporabimo pri metodi `MessageBox.Show`. Levo zgoraj je informacija, koliko različnih oblik izbrane metode obstaja. V našem primeru jih je 12. S klikom na puščici gor ali dol se sprehajamo med vsemi metodami, kjer je prikazana glava metode, pod njo pa opis metode in označenega parametra.

Predelovanje kode (Refactoring)

Pri razvoju nam je v veliko pomoč možnost preimenovanja posameznih spremenljivk, objektov, metod kot tudi to, da del kode izločimo v samostojno metodo, in podobno. Pri tem si pomagamo z orodjem za predelovanje kode.

S pomočjo tega orodja zlahka zamenjamo na primer ime določene metode ali pa spremenljivke. Orodje poskrbi, da se bo ime spremenilo povsod, kjer je to potrebno. To pomeni, da bo ime zamenjano v vseh datotekah, ki sestavljajo projekt, na vseh mestih, kjer metodo kličemo … Pri tem se ne more zgoditi, da bi s tem morda zamenjali tudi klic enako poimenovane funkcije iz drugega razreda. Navadni tekstovni urejevalnik pri "Poišči in zamenjaj" naredi prav to, saj je tam pomembno le ujemanje zaporedja znakov in kontekst ni upoštevan.

Poleg sprememb imen je priročna tudi možnost, da iz določenega dela kode lahko naredimo metodo. Denimo da opazimo, da smo večkrat zapisali enak ali podoben del kode, ali pa ugotovimo, da bi morali ponovno zapisati del kode, ki smo jo natipkali že prej. Takrat je priročno, da ustrezni del kode na preprost način pretvorimo v metodo, ki jo potem kličemo na ustreznih mestih.



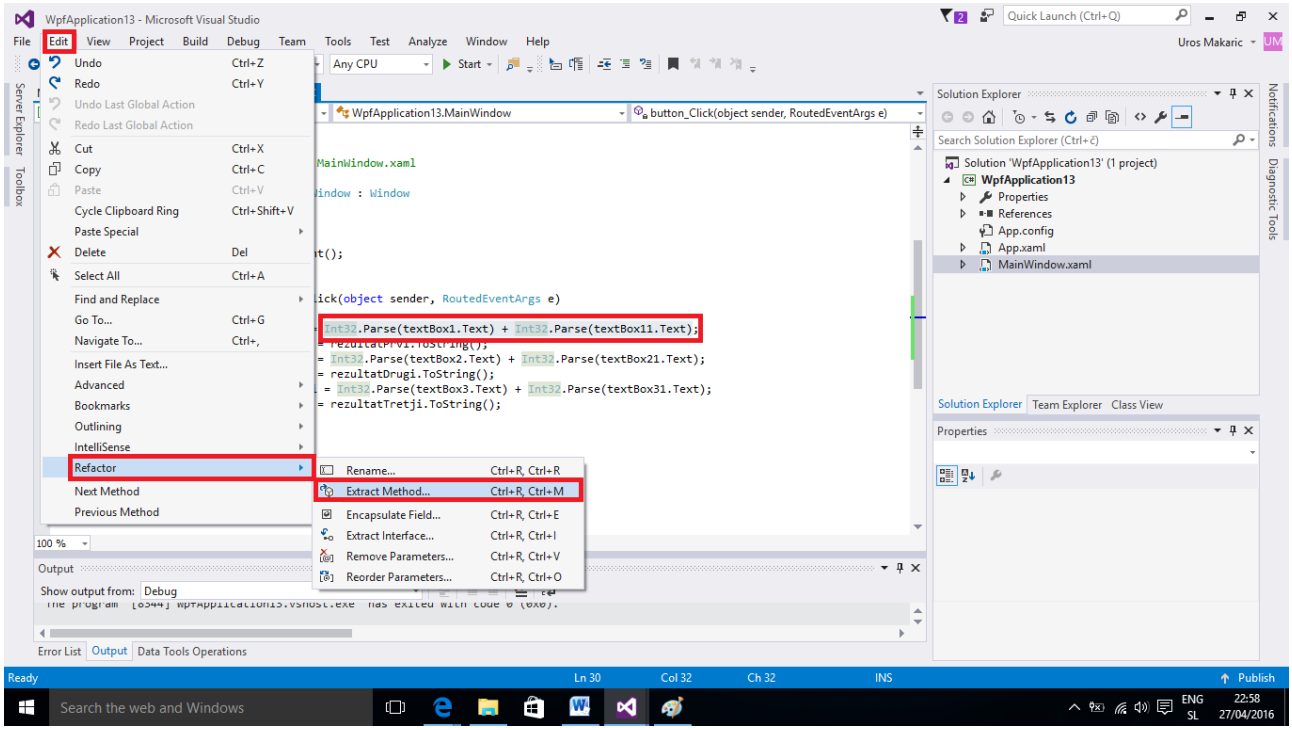
Slika 26: Aplikacija, ki računa seštevek dveh številk znotraj besedilnih polj

Oglejmo si primer, kjer sestavljamo aplikacijo, kamor uporabnik v šest vnosnih polj vnese številke, s klikom na gumb pa aplikacija izračuna seštevek cifer vnosnih polj (glej sliko 26).

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void button_Click(object sender, RoutedEventArgs e)
    {
        int rezultatPrvi = Int32.Parse(textBox1.Text) + Int32.Parse(textBox11.Text);
        rezultat1.Content = rezultatPrvi.ToString();
        int rezultatDrugi = Int32.Parse(textBox2.Text) + Int32.Parse(textBox21.Text);
        rezultat2.Content = rezultatDrugi.ToString();
        int rezultatTretji = Int32.Parse(textBox3.Text) + Int32.Parse(textBox31.Text);
        rezultat3.Content = rezultatTretji.ToString();
    }
}
```

Slika 27: Prvotna koda, ki izračuna vrednosti in jih zapiše

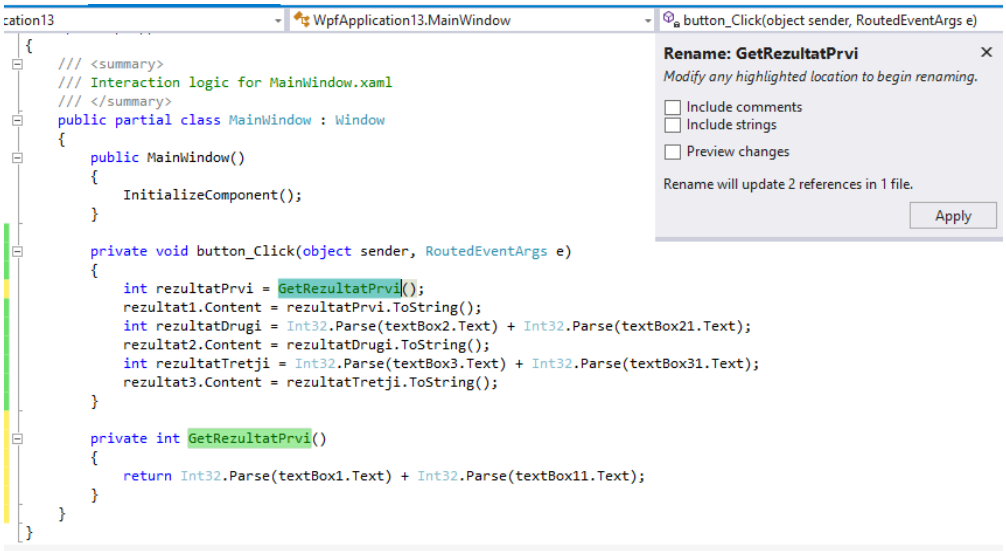
Na sliki 27 vidimo kodo, ki nam vrne seštevke dveh številk. Iz slike vidimo, da se koda, ki izračuna seštevek dveh besedilnih polj, trikrat ponovi. Da ne bi pisali kode še enkrat v primeru, ko bi dodali še dve besedilni polji, ali pa želimo kodo urediti, lahko ta del preuredimo z izločanjem v funkcijo.



Slika 28: Primer dela kode, ki jo preuredimo v izločanje v funkcijo

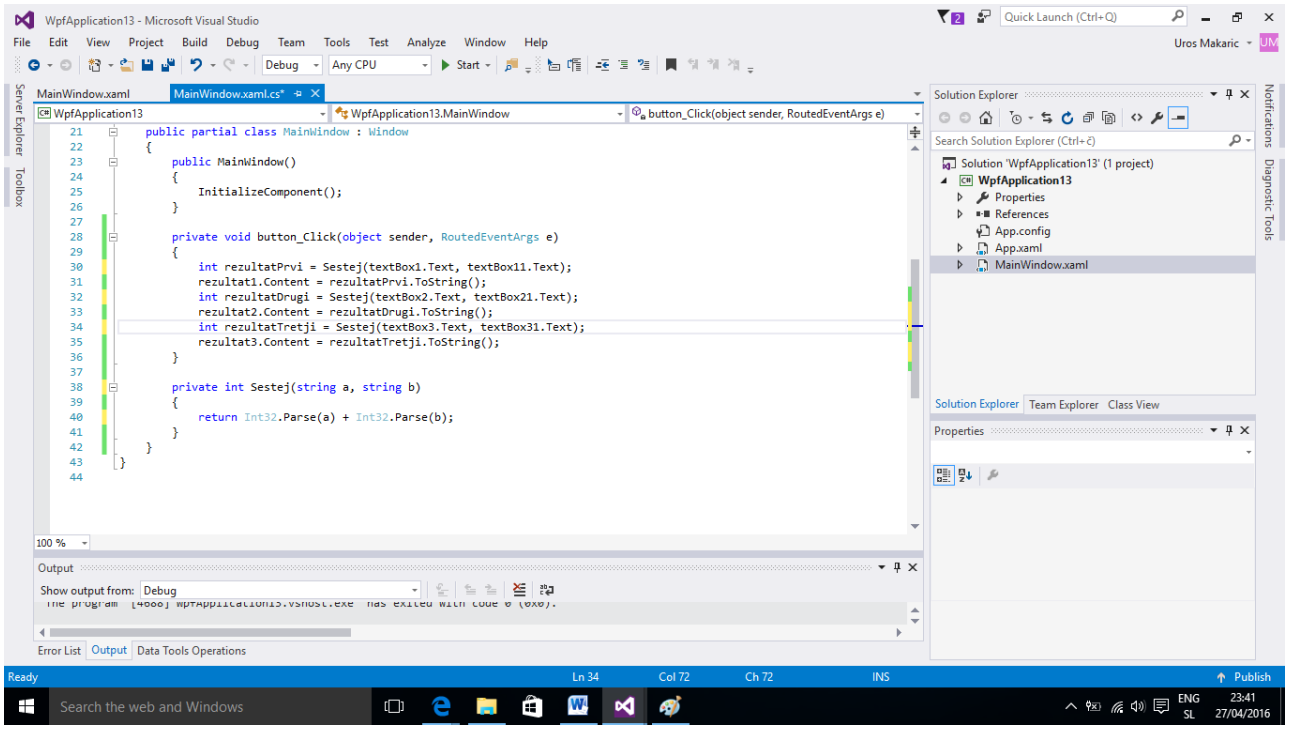
To storimo tako, da označimo del kode, kjer je zapisano, da se sešteje besedilo v `textBox1.Text` in `textBox11.Text`. Nato v menijski vrstici Visual Studia izberemo `Edit` in v spustnem meniju poiščemo `Refactor`. S klikom na trikotnik se nam razširi izbor, kjer kliknemo `Extract Method ...` (glej označene dele slike 28).

Po kliku se nam v urejevalniku koda `Int32.Parse(textBox1.Text) + Int32.Parse(textBox11.Text)` spremeni v `GetRezultatPrvi()`. Dobimo novo funkcijo s podpisom `private int GetRezultatPrvi()`, ki nam vrne seštevek števk, zapisanih v poljih `textBox1` in `textBox11`. Na sliki 29 v desnem zgornjem kotu urejevalnika kode vidimo, da nam okolje med postopkom ponudi možnost preimenovanja funkcije.



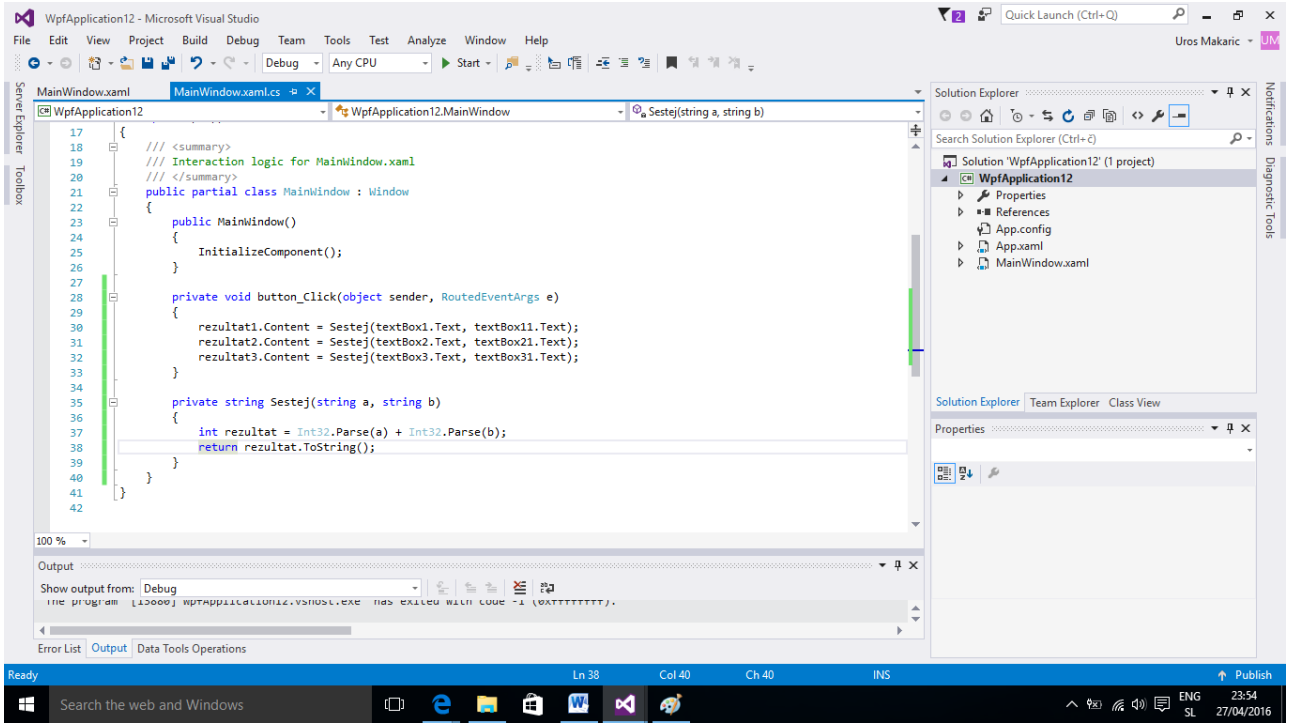
Slika 29: Poimenovanje funkcije, ki jo ustvarimo z izločanjem

Seveda postopek ne zna sam opraviti vseh smiselnih sprememb. Tako sami dodamo parametra a in b.



Slika 30: Funkciji Sestej () dodamo dva parametra in jo uporabimo za več besedilnih polj

S tem je funkcijo možno uporabiti na več mestih (slika 30).



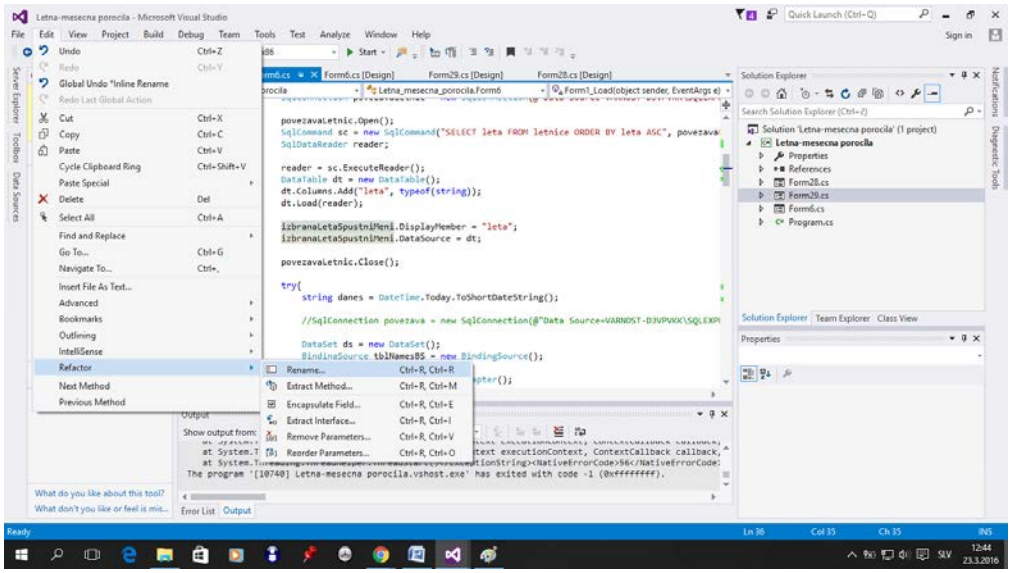
Slika 31: Sprememba funkcije Sestej(), ki je vračala cela števila v besedilo

To storimo tako, da spremenimo, katere vrednosti bo funkcija Sestej() vračala. Ker aplikacija ob kliku na gumb Izračunaj vrne besedilni zapis številke, je potrebno, da funkcijo `private int Sestej(string a, string b)` spremenimo v `private string Sestej(string a, string b)`. Tako nam funkcija ne bo vračala številke, ampak besedilni zapis le-te.

Končna različica začetne kode (slika 27) je bolj urejena in pregledna ter kot taka bolj razumljiva (glej sliko 32).

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void button_Click(object sender, RoutedEventArgs e)
    {
        rezultat1.Content = Sestej(textBox1.Text, textBox11.Text);
        rezultat2.Content = Sestej(textBox2.Text, textBox21.Text);
        rezultat3.Content = Sestej(textBox3.Text, textBox31.Text);
    }

    private string Sestej(string a, string b)
    {
        int rezultat = Int32.Parse(a) + Int32.Parse(b);
        return rezultat.ToString();
    }
```

Slika 32: Preurejena koda



Slika 33: Primer spremembe imena spremenljivke

Poleg izločanja v funkcijo lahko funkcije in spremenljivke tudi preimenujemo. Na sliki 33 vidimo, kako lahko na preprost način spremenimo ime določene spremenljivke. Najprej z dvojnim klikom označimo ime spremenljivke (na poljubnem mestu v kodi). Potem v menijski vrstici kliknemo na `Edit`, izberemo možnost `Refactor` in nato `Rename`. V izbirnem oknu vnesemo novo ime. Pri tem se nam ponudi možnost, da lahko to ime zamenjamo tudi v komentiranih delih in besedilnih nizih.

Označevanje in skrivanje območij

Pri pisanju obsežnejših aplikacij je zelo dobro, da imamo lahko kar se da celovit pregled nad kodo. Pri tem nam pomagajo možnost skrivanja vsebine funkcij in zaznamki (`Bookmark`), ki smo jih že omenili v razdelku Urejevalnik kode.

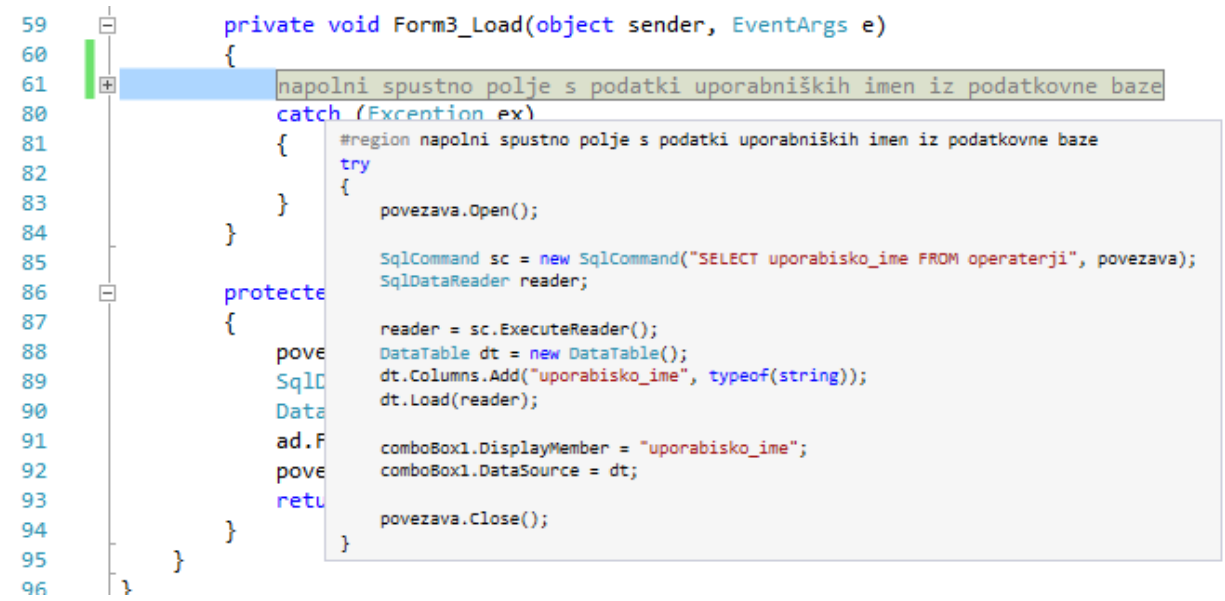
Kot smo omenili že v splošnem opisu urejevalnika kode, lahko s klikom na znak –, skrijemo vsebino funkcije oziroma z znakom + prikažemo kodo, ki sestavlja funkcijo. Pri razvoju obsežnih aplikacij se lahko zgodi, da to ni dovolj, saj je lahko tudi koda znotraj funkcije zelo obsežna. Takrat si lahko pomagamo tako, da določimo posamezna območja kode oziroma regije.

Območje dobimo tako, da na začetku želenega območja napišemo `#region` in na koncu `#endregion`. Tako označeno območje (del kode) lahko poljubno prikažemo ali skrijemo. Ko je območje skrito, vidimo le oznako za začetek kode. Zato je smiselno, da za `#region` napišemo še komentar, ki pove, kaj vsebuje to območje. Kot vidimo na sliki 35, ko gremo z miško čez napis `#region`, Visual Studio prikaže kodo znotraj skritega bloka enako, kot takrat, ko je skrita vsebina funkcije.

```
59      private void Form3_Load(object sender, EventArgs e)
60      {
61          #region napolni spustno polje s podatki uporabniških imen iz podatkovne baze
62          try
63          {
64              povezava.Open();
65
66              SqlCommand sc = new SqlCommand("SELECT uporabisko_ime FROM operaterji", povezava);
67              SqlDataReader reader;
68
69              reader = sc.ExecuteReader();
70              DataTable dt = new DataTable();
71              dt.Columns.Add("uporabisko_ime", typeof(string));
72              dt.Load(reader);
73
74              comboBox1.DisplayMember = "uporabisko_ime";
75              comboBox1.DataSource = dt;
76
77              povezava.Close();
78          }
79          #endregion
80          catch (Exception ex)
81          {
82              MessageBox.Show(ex.Message);
83          }
84      }
```

```
59      private void Form3_Load(object sender, EventArgs e)
60      {
61          napolni spustno polje s podatki uporabniških imen iz podatkovne baze
80          catch (Exception ex)
81          {
82              MessageBox.Show(ex.Message);
83          }
84      }
```

Slika 34: Zgornji del slike 34, ko je območje razširjeno, in spodnji, ko ni (glej številke vrstic)

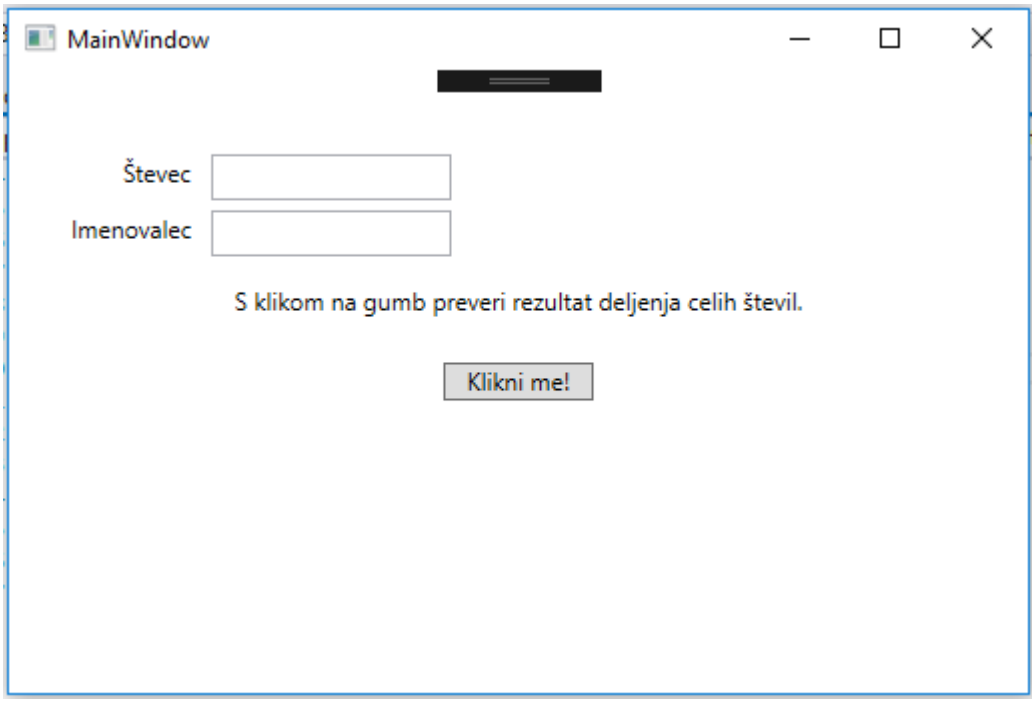


Slika 35: Ko gre razvijalec z miško čez komentar, se prikaže koda znotraj območja – region

Razhroščevalnik (Debugger)

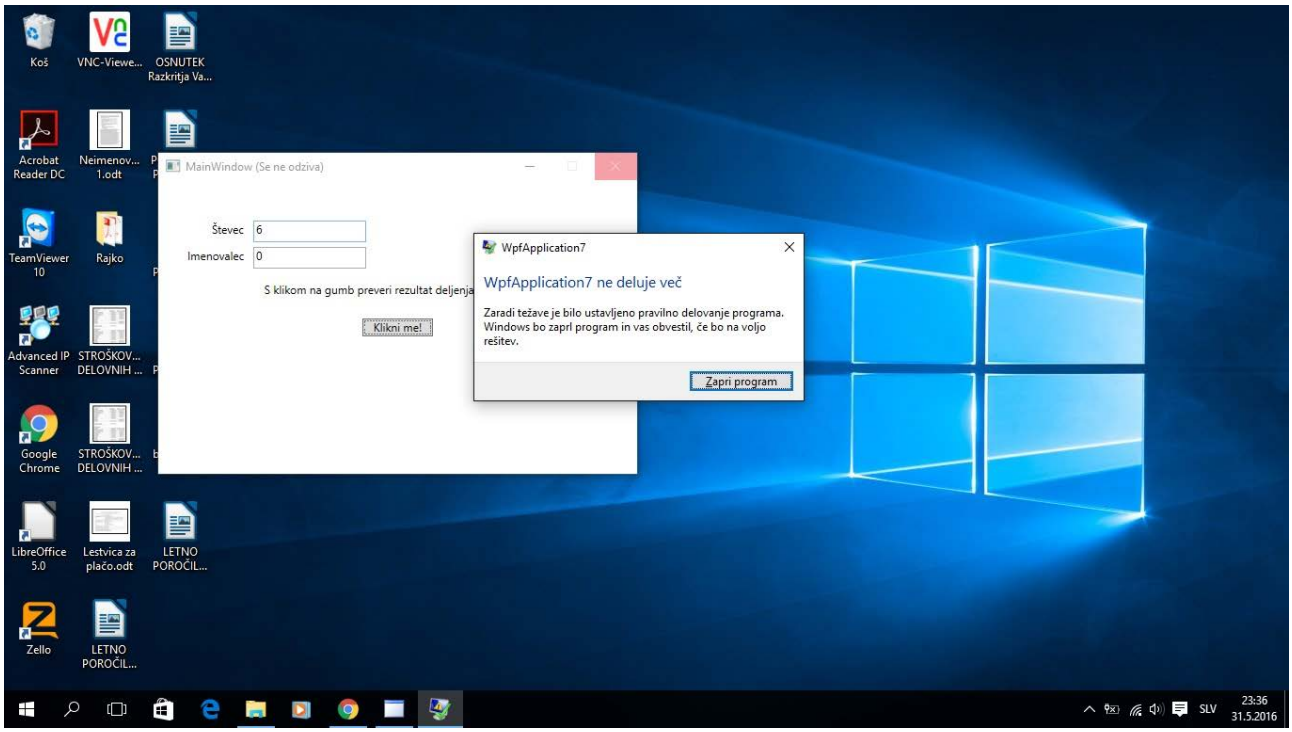
Ko napišemo kodo in odpravimo sintaktične napake, se začne težavno delo – odpravljanje semantičnih (pomenskih napak). To so napake, ko sintaktično pravilni program ne vrne pričakovanih (pravilnih) rezultatov. Pri odpravljanju tovrstnih napak si lahko pomagamo z orodjem, ki mu rečemo razhroščevalnik. Ta ponuja različne prijeme, ki nam pomagajo odkriti te napake. V okolju VS je razhroščevalnik zelo zmogljiv. Ponuja številne možnosti, ki razvijalcu pomagajo, da lažje odkrije in odpravi napake. Mi si bomo ogledali le najbolj osnovno uporabo. Več o uporabi razhroščevalnika in njegovi naprednejši uporabi si lahko preberete na primer v [17], [18], [19] in [20].

Oglejmo si najprej preprost primer. Na sliki 36 vidimo program, ki preveri, ali sta števili, ki ju vstavimo v besedilni polji, deljivi.



Slika 36: Program, ki preveri, ali sta dve vpisani števili deljivi

S klikom na gumb `"Klikni me!"` dobimo odgovor, ali je število v polju Števec deljivo s številom v polju Imenovalec. Ko zaženemo program in v polje Števec vstavimo npr. 6 in v polje Imenovalec 2, se program izvede in na okno izpiše besedilo `"Število 6 je deljivo z 2."` Ker smo s preizkusom zadovoljni, program ustrezno zapakiramo (o tem bo govora v razdelku Objava aplikacij za namestitev) in ga predamo naročniku. Ko pa naročnik začne uporabljati program, vidi prizor, kot je na sliki 37.



Slika 37: Opozorilo, ki nam pove, da se je aplikacija zaustavila

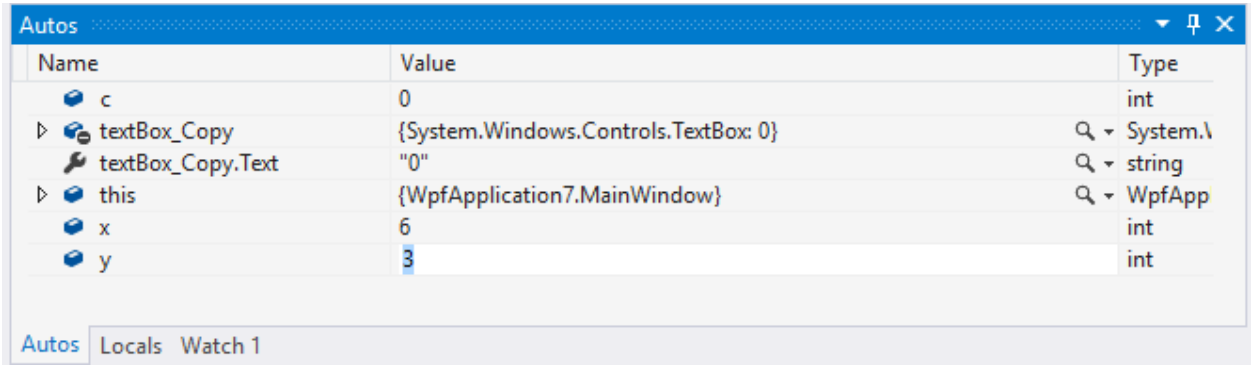
Med izvajanjem programa je prišlo do napake. Očitno programa pred predajo v uporabo nismo ustrezno testirali.

Poiskati moramo torej napako v programu. Z obvestilom, ki ga je prejel uporabnik, si ne moremo kaj dosti pomagati. Na srečo pa nam, če pride do napak pri preizkušanju programa v samem okolju VS, okolje pomaga z ustreznimi opozorili.

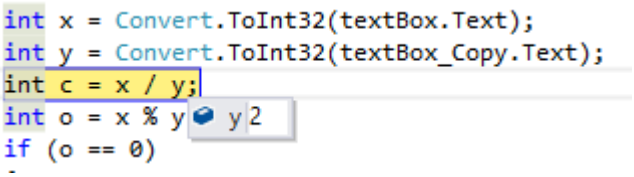
Ker nam je uporabnik povedal, da je do napake prišlo, ko je kliknil na gumb `"Klikni me!"` in je bilo stanje tako, kot je na sliki 37, poskusimo to izvesti tudi sami. Kot števec vnesemo število 6 in za imenovalec 0. Ob kliku na gumb se program (pričakovano) ustavi, vendar nas zdaj vrne v urejevalnik kode. Pojavi se okno, ki nas opozori, da je prišlo do izjeme, ki ni bila upoštevana v vrstici 32 (glej sliko 40). V tem oknu so podane dodatne informacije, ki nam povedo, kako lahko odpravimo to napako. Tako nam v našem primeru izpiše opozorilo `"DivideByZeroException was unhandled"` in nam v oknu ponudi povezavi do spletne strani s podrobnejšo razlago, za kateri predvideva, da bi nam pomagali pri rešitvi težave. Ob tem nam izpiše še opozorilo, da je program hotel deliti s številom 0 (`"Attempted to divide by zero."`).

Na sliki 40 levo spodaj vidimo zavihek `Autos`. V njem vidimo, kakšne so vrednosti spremenljivk v trenutku, ko se je program ustavil. Če želimo, lahko vrednosti teh spremenljivk spremenimo in nadaljujemo z izvajanjem programa. Na ta način enostavno preverimo, ali je težava le v tem, da so vrednosti določenih spremenljivk napačne. Tako lahko spremenimo vrednost spremenljivke `y` in namesto 0 v stolpcu `Value` vpišemo drugo vrednost. Po spremembi nadaljujemo s programom tako, da kliknemo na gumb `Continue`, ali pa uporabimo tipko `F5`. Če pogledamo na sliko 38, vidimo, da smo vrednost spremenljivke `y` spremenili na 3. Poleg tega lahko spremenimo vrednost spremenljivke tudi tako, da se v prekinitvenem načinu z miško

postavimo na poljubno spremenljivko v kodi. Pod njo se nam v oknu pokaže njena vrednost, ki jo lahko nato spreminjamo (glej sliko 39).

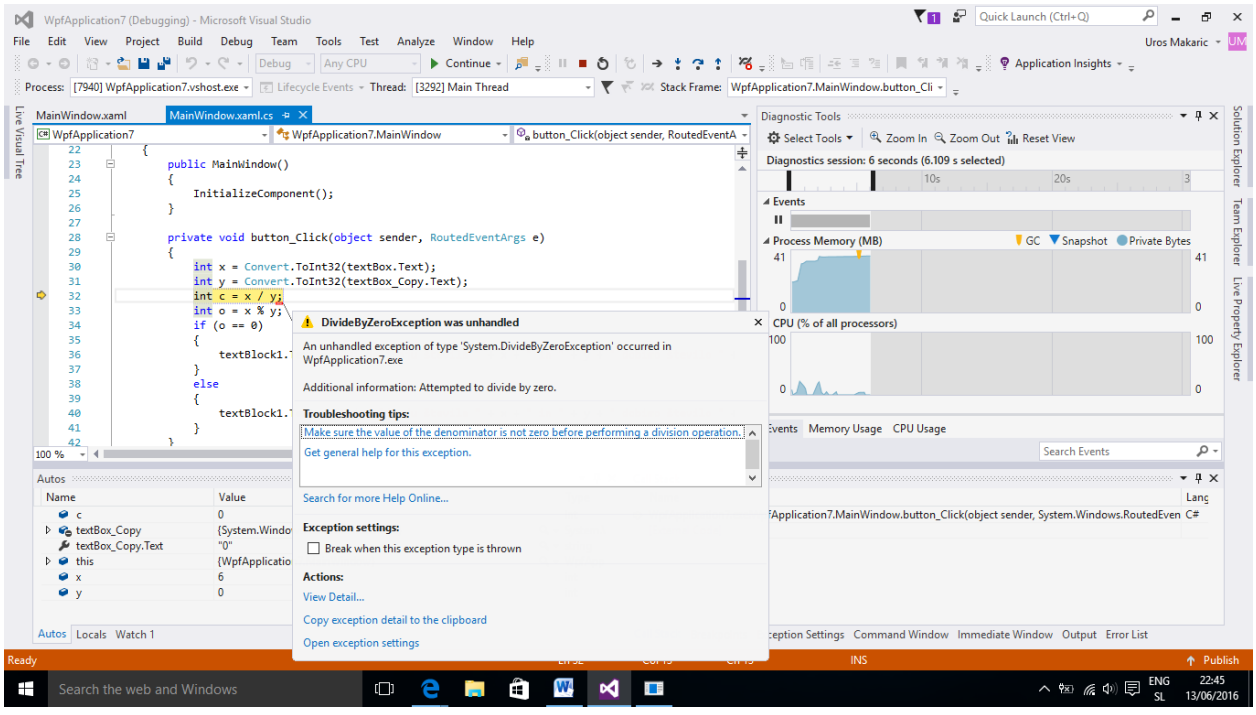


Slika 38: Sprememba vrednosti spremenljivke y v zavihku Autos



Slika 39: Sprememba vrednosti spremenljivke y v kodi

Poleg zavihka `Autos` imamo tudi zavihek `Locals`. Zavihek `Locals` ima podobno funkcijo kot `Autos` s to razliko, da so v zavihku `Locals` prikazane vse lokalne spremenljivke znotraj bloka, kjer se je program zaustavil.



Slika 40: Primer napake, ko v polje imenovalec vpišemo število 0

V našem primeru pri pregledu kode hitro vidimo, da smo pozabili preveriti možnost deljenja z 0, ki bi uporabnika opozorila, da deljenje z 0 ni dovoljeno.

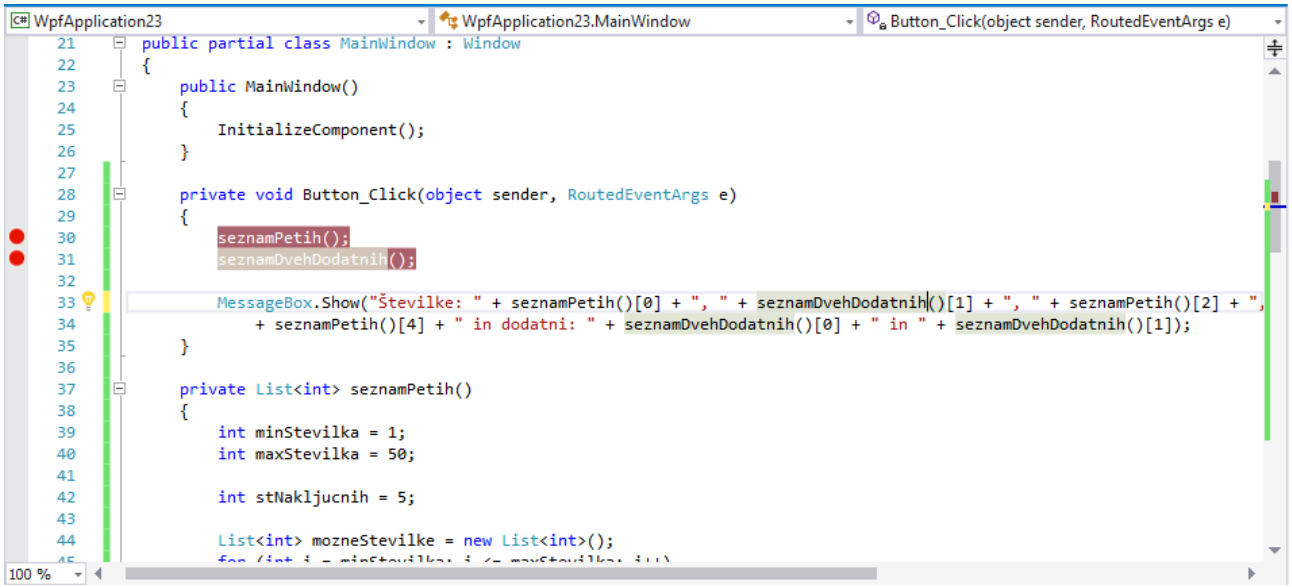
V predstavljenem primeru smo napako hitro odkrili in odpravili. Pogosto pa ne vemo, kaj je bil vzrok napake, zakaj je prišlo do težav in podobno. Takrat si pomagamo s prekinitvenimi točkami. To so točke, ki jih postavimo na začetek tistih delov kode, kjer sumimo, da je napaka. Potem, ko zaženemo program, se na

teh mestih program ustavi. Zato lahko s pomočjo oken `Autos` in `Locals` dobimo informacije o trenutnih vrednostih spremenljivk. Iz teh informacij lahko nato sklepamo, ali koda, ki se izvaja, pravilno opravlja svoje delo. Kot smo pokazali že prej, po potrebi vrednosti določenih spremenljivk spremenimo kar ročno in nadaljujemo z izvajanjem programa. Ta se izvaja do naslednje prekinitvene točke. Kot bomo videli v nadaljevanju, pa po vsaki taki prekinitvi izvajanja, program lahko izvajamo tudi korak za korakom.

Prekinitvene točke nastavimo vnaprej, še preden program zaženemo. A ena izmed zelo uporabnih stvari v razhroščevalniku VS je ta, da lahko prekinitvene točke dodamo in poljubno nastavljamo tudi potem, ko že zaženemo razhroščevalnik.

Poglejmo si zgled, kjer bomo uporabili prekinitvene točke. Na sliki 41vidimo del kode programa, ki nam vrne naključne številke za igranje igre `EuroJackpot`. Za aplikacijo pričakujemo, da nam bo vrnila 7 različnih naključnih števil, kjer bo 5 števil med 1 in 50 ter 2 števili med 1 in 9. Ker nam program občasno vrne napačne rezultate, si pomagamo z razhroščevanjem. Pri tem sumimo, da se napaka pojavi v metodah, ki nam vrnejo naključna števila, zato postavimo prekinitvene točke na ta mesta. To storimo tako, da najprej izberemo vrstico, kamor želimo postaviti prekinitveno točko. To postavimo tako, da kliknemo z miško na levi rob vrstice s kodo. Drugi način pa je, da iz menijske vrstice izberemo `Debug` in `Toggle Breakpoint,` ali pa pritisnemo tipko `F9`.

Na levi strani pred številkami vrstic kode vidimo rdeče pike. Te pike so prekinitvene točke. Ko pri izvajanju pridemo do teh točk, se delovanje programa začasno prekine. Zato lahko preverimo takratno vrednost spremenljivk. Na sliki 42 sta označeni dve prekinitveni točki, v zavihku `Locals` (spodaj levo slika 42) pa vidimo, kakšne so trenutne vrednosti spremenljivk `minStevilka`, `maxStevilka`, `stNakljucnih`, `mozneStevilke`, `seznam`, `nakljucna` in `i`.

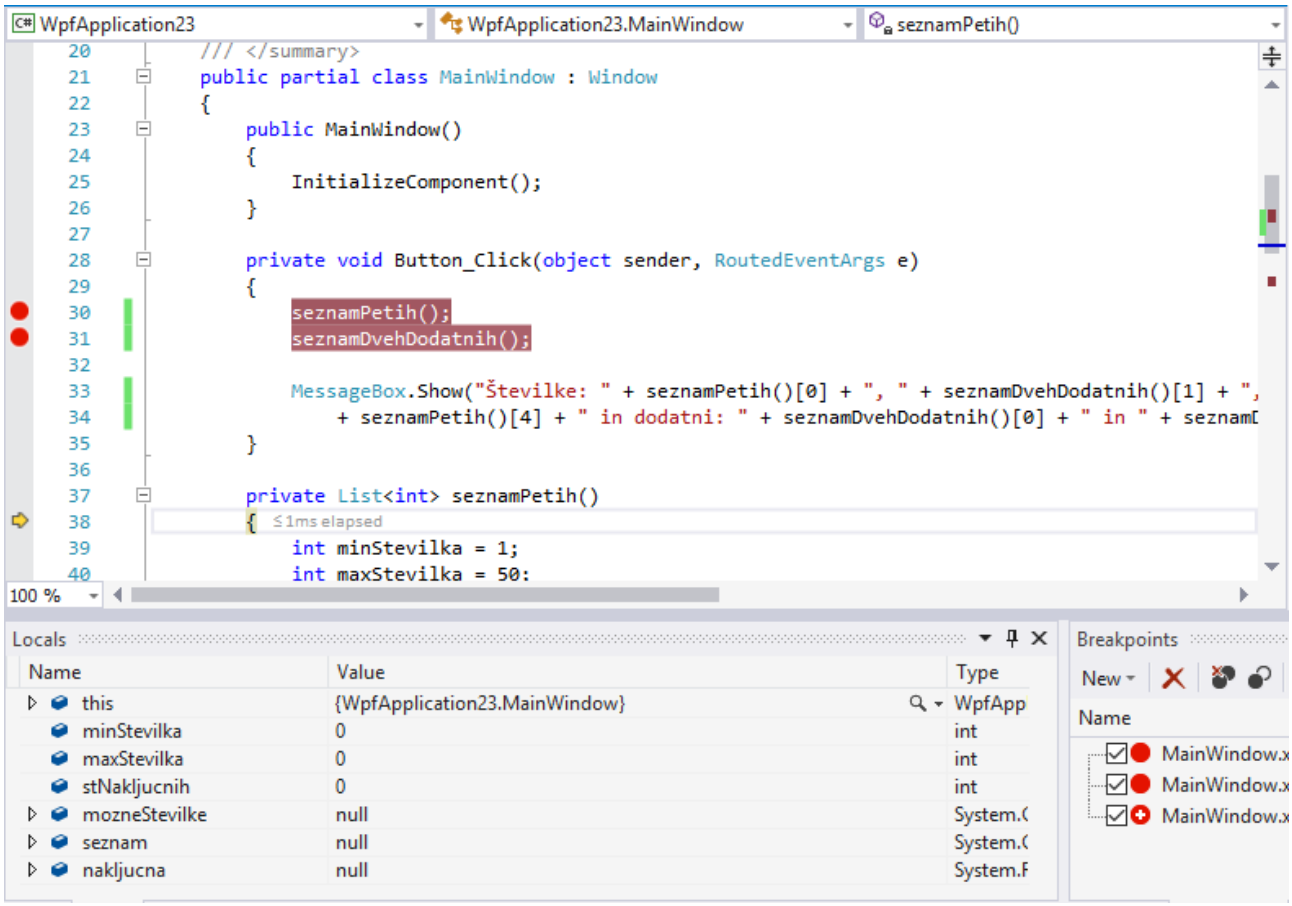


Slika 41: Prekinitveni točki v vrsticah 30 in 31

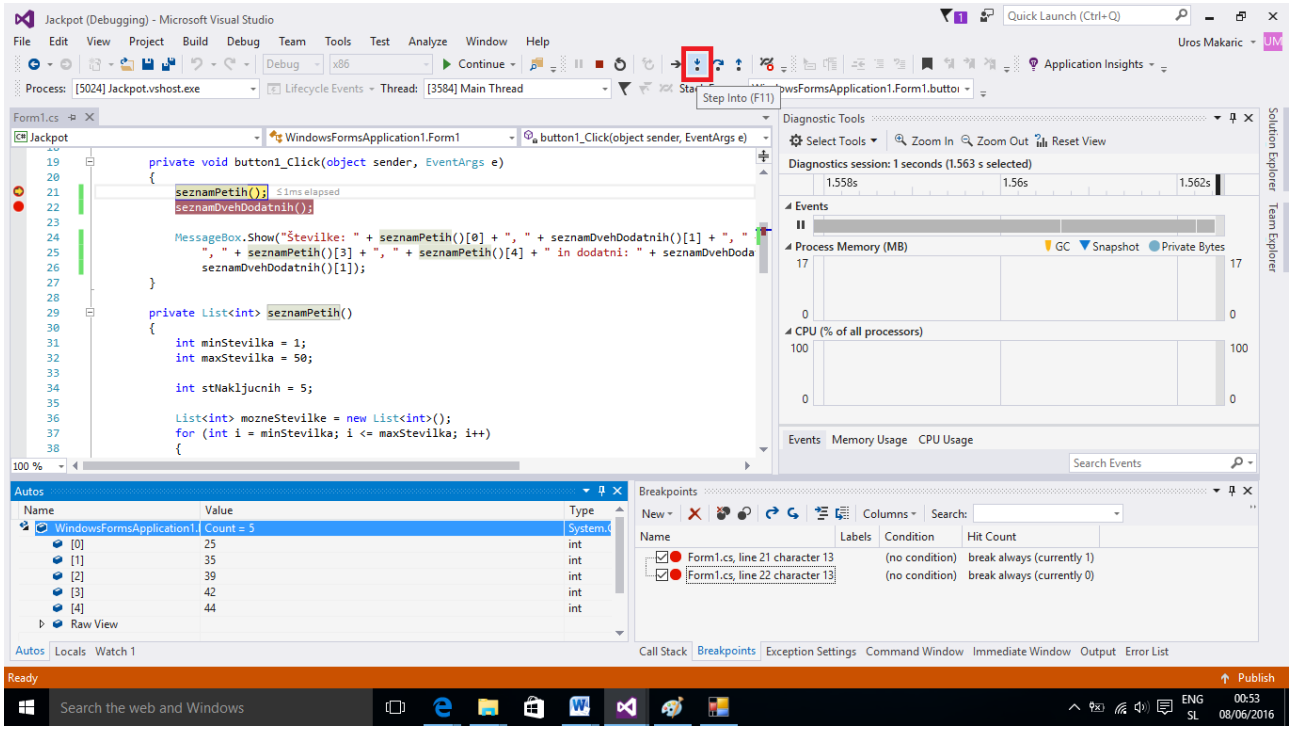
Ker pa v našem primeru z uporabo prekinitvenih točk nismo ugotovili vzroka napake, si pomagamo s koračnim izvajanjem programa. Koračno izvajanje pomeni, da se bo program izvajal korak za korakom. S takim načinom izvajanja lahko začnemo že takoj na začetku. Bolj običajno pa je, da pustimo, da se program izvaja do prekinitvene točke. Od tam dalje kodo izvajamo koračno. Z ukazi `Step Into` (bližnjica je tipka `F11`), `Step Over` (`F10`) in `Step Out` (`Shift + F11`) se premikamo po vrsticah kode in tako sproti preverjamo vrednosti spremenljivk. Pri tem lahko že iz imena ukaza sklepamo, kaj se zgodi, če izberemo npr. `Step Into`. S tem vstopimo v kodo klicane metode in začnemo s preverjanjem dogajanja znotraj metode. Če kaže, da se koda metode obnaša pravilno, jo zapustimo (torej izvedemo vse preostale ukaze znotraj

metode) z ukazom `Step Out`. S pomočjo `Step Over` se v enem koraku premaknemo preko vsake vrstice, tudi če ta vsebuje klice metod.

Če v vrstici, ki je na vrsti za izvajanje, ni klica nobene metode, je povsem vseeno, ali uporabimo `Step Into` ali `Step Over`.

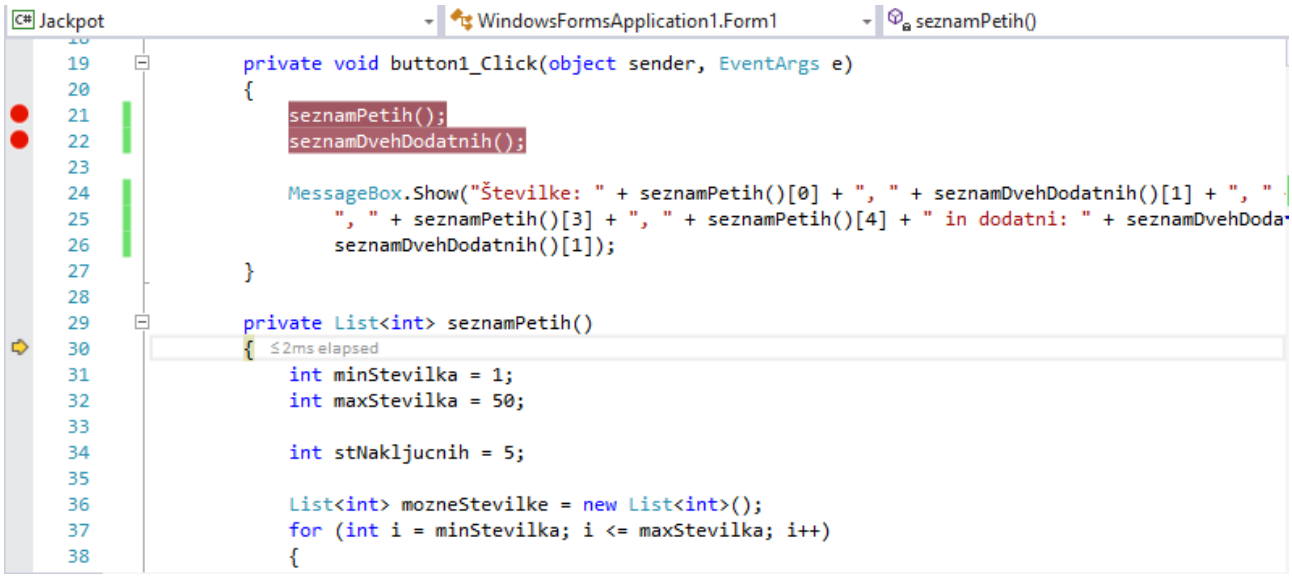


Slika 42: Uporaba koračnega izvajanja in spremljanje vrednosti spremenljivk



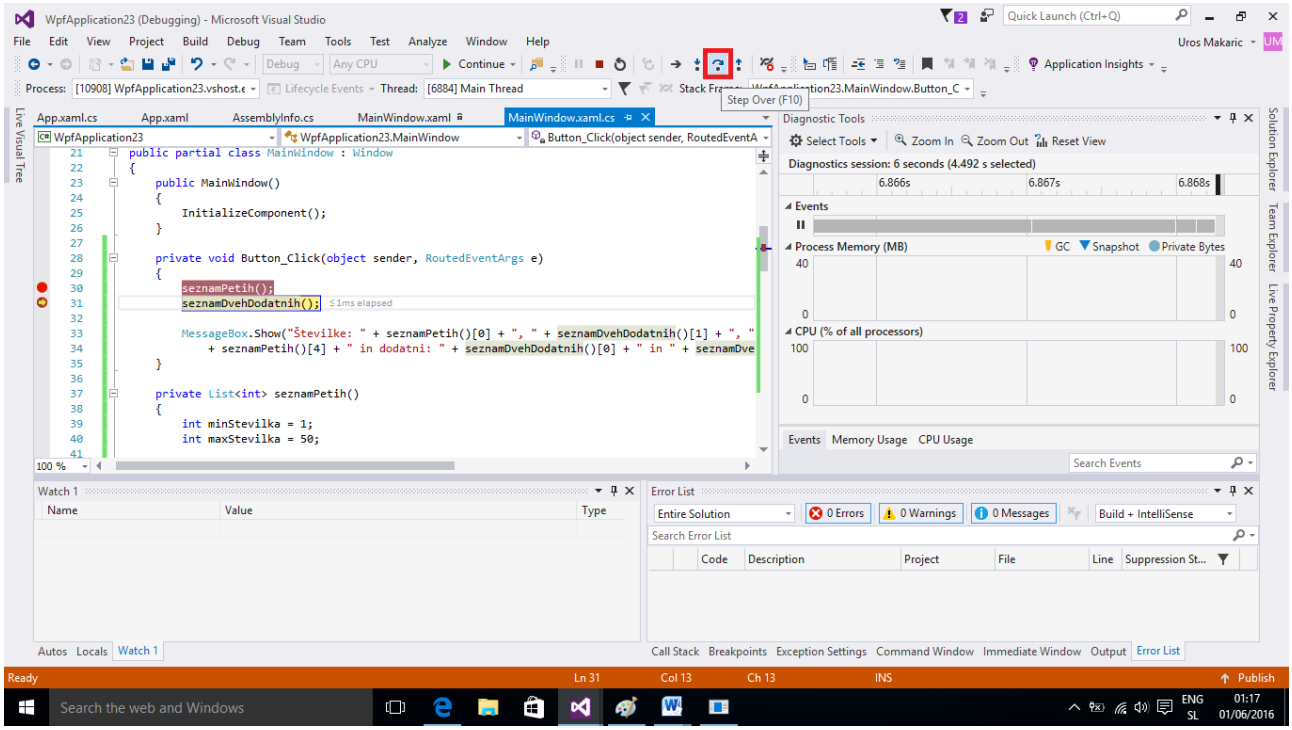
Slika 43: Koračno izvajanje s pomočjo tipk Step Into, Step Over in Step Out

Ko zaženemo program v VS, se program ustavi na prvi prekinitveni točki, ko bi morala biti klicana metoda `seznamPetih()`. Z ukazom `Step Into` vstopimo v metodo. Ustavimo se na prvi vrstici metode (slika 44). Rumena puščica nam namreč označuje vrstico, ki bo izvedena v naslednjem koraku.



Slika 44: Vstop v metodo z ukazom Step Into

V zavihku `Autos` preverimo vrednosti spremenljivk (slika 42). Ker so vrednosti spremenljivk na tem delu pričakovane, se s klikom na gumb `Step Into` (rdeči kvadrat na sliki 43) premaknemo v naslednjo vrstico kode. Tu preverimo, ali je prva vrstica v metodi `seznamPetih()` smiselno opravila delo. Tako se premikamo in preverjamo vrednosti spremenljivk korak za korakom. Če ugotovimo, da je metoda od nekega dela naprej zagotovo pravilna, lahko izvedemo vse ukaze do konca metode tako, da uporabimo gumb `Step Out` (gumb desno od gumba `Step Over` na sliki 45) in se vrnemo iz te metode na mesto za njenim klicem.



Slika 45: Uporaba tipke Step Over pri razhroščevanju

Če vemo, da metoda dela pravilno, lahko s klikom na `Step Over` izvedemo celotno metodo v enem koraku in se premaknemo na naslednjo vrstico.

```
private void button1_Click(object sender, EventArgs e)
{
    seznamPetih();
    seznamDvehDodatnih();

    MessageBox.Show("Številke: " + seznamPetih()[0] + ", " + seznamDvehDodatnih()[1] + ", " + seznamPetih()[2] +
        ", " + seznamPetih()[3] + ", " + seznamPetih()[4] + " in dodatni: " + seznamDvehDodatnih()[0] + " in " +
        seznamDvehDodatnih()[1]);
}
```

Slika 46: Izpis sedmih naključnih števil programa

Tako s koračnim izvajanjem ugotovimo, da smo napravili napako pri izpisu besedila na okno. Namesto števila iz seznama `seznamPetih()` smo se zatipkali in nam program izpiše število iz seznama `seznamDvehDodatnih()` (svetlo modro označen del na sliki 46). Poleg tega pa ugotovimo, da se metodi `seznamPetih()` in `seznamDvehDodatnih()` ne kličeta le enkrat. Zaradi te napake smo občasno dobivali v izpisu dve ali več enakih števil s `seznamaPetih()`. To popravimo tako, da ustvarimo npr. spremenljivko `a`, v katero shranimo seznam petih naključnih številk s klicem metode `seznamPetih()`. Podobno naredimo z metodo `seznamDodatnihDveh()`, ki jo shranimo v spremenljivko `b` (slika 47).

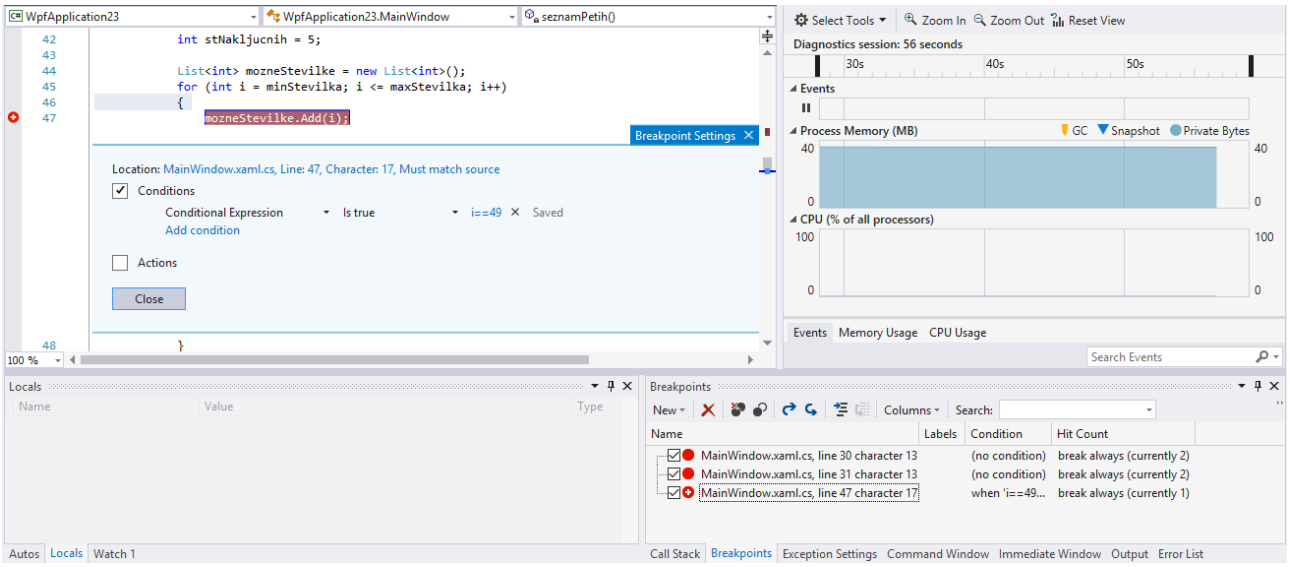
```
private void Button_Click(object sender, RoutedEventArgs e)
{
    List<int> a = seznamPetih();
    List<int> b = seznamDvehDodatnih();

    MessageBox.Show("Številke: " + a[0] + ", " + a[1] + ", " + a[2] + ", " + a[3] + ", "
        + a[4] + " in dodatni: " + b[0] + " in " + b[1]);
}
```

Slika 47: Urejena koda programa po odpravljenih napakah s pomočjo koračnega izvajanja

Na prikazanem primeru smo predstavili osnovni način uporabe razhroščevalnika. Omenimo še nekaj njegovih značilnosti, ki nam utegnejo priti prav.

Na desni strani spodaj imamo ob razhroščevanju za pomoč pri delu več zavihkov. Med njimi sta najbolj uporabna `Call Stack` in `Breakpoints`. V zavihku `Breakpoints` vidimo (slika 48), kam smo postavili prekinitvene točke. Tam lahko te prekinitvene točke urejamo, jih poimenujemo, dodajamo pogoje, kdaj se upoštevajo … Zavihek `Call Stack` nam pokaže sklad klicanih metod na določenem mestu v prekinitvenem načinu, ki še čakajo na zaključek izvedbe. Iz tega lahko hitro ugotovimo, katere metode so povezane med seboj.

Kot smo omenili, lahko prekinitvenim točkam dodajamo tudi pogoje. S tem povemo, da naj se izvajanje kode na tem mestu prekine le, če je izpolnjen določen pogoj. Pogoj dodamo tako, da v zavihku `Breakpoints` kliknemo na `New` in izberemo možnost `Function Breakpoint`. Potem se nam pojavi okno, v katerem določimo, da se na tem mestu program ustavi ob določenem pogoju. Na sliki 48 smo določili, da se program v 47. vrstici kode ustavi le takrat, ko je vrednost spremenljivke i enaka 49. Pogojne prekinitvene točke nam pridejo prav predvsem takrat, ko imamo dolge zanke in sumimo, da so težave le pri določeni ponovitvi te zanke. Če bi morali vse korake zanke izvajati s `Step Into` (ali `Step Over`), bi nam to vzelo veliko časa. Tako pa lahko pustimo, da se večina zanke izvede hitro, program pa se ustavi, ko dosežemo sumljivo ponovitev.



Slika 48: Pregled zavihkov Breakpoints

Raziskovalec rešitve (Solution Explorer)

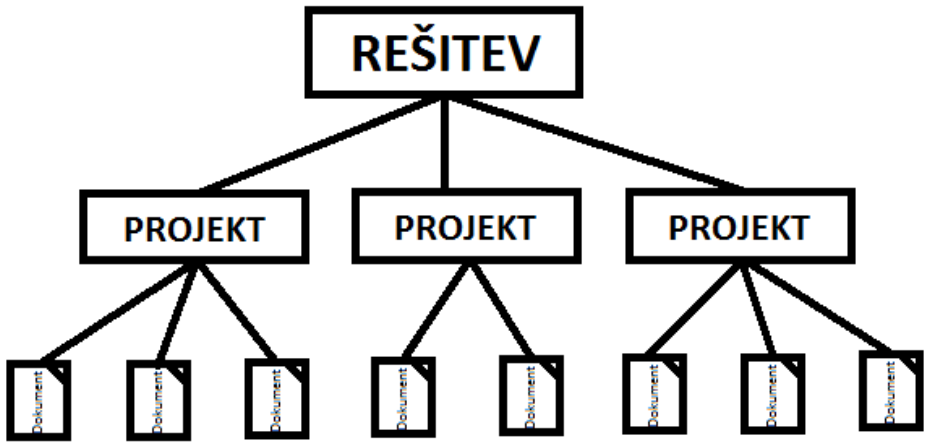
Kot smo videli že v razdelku Izdelava projektov, vedno začnemo z izbiro projekta. Projekt sestavljajo različne datoteke (datoteke s kodo projekta, ikone, podatkovne datoteke, slike …), ki jih uporabimo za izdelavo aplikacije. V projekt so vključene tudi druge datoteke, ki jih potrebujemo pri storitvah ali zunanjih komponentah, s katerimi je aplikacija povezana. Vse te informacije so shranjene v posebno projektno datoteko. Ta datoteka (`.csproj` za C# projekte) torej vsebuje prikaz nastavitev in vsebine projekta.

Pri reševanju obsežnejših problemov pa običajno potrebujemo več projektov, ki so med seboj povezani in uporabljajo tudi določene skupne vire (npr. imajo skupne grafične predloge, podatkovne baze …).

V VS tak skupek med seboj odvisnih projektov imenujemo rešitev.

Rešitev je organizacijska enota, s katero v razvojnem okolju VS upravljamo enega ali več projektov. Sestavlja jo lahko več posameznih projektov, ki so pisani v različnih programskih jezikih (takih, ki jih podpira VS

Community). VS za shranjevanje nastavitev uporablja dve vrsti datotek (`.sln` in `.suo`). Ti dve datoteki izdela raziskovalec rešitve in ju opremi z vsemi informacijami ter podatki. Datoteka `.sln` (Visual Studio Solution) vsebuje popoln opis o vsebini rešitve. Ta datoteka organizira projekt in njegove komponente v skupno rešitev, ki jo lahko uporablja več razvijalcev. Datoteka `.suo` (Solution User Options) pa vsebuje podatke o položaju oken. Datoteka je specifična za posameznega razvijalca, hrani pa vse potrebne nastavitve razvojnega okolja, zato da jih pri ponovnem odpiranju projekta ni treba nastavljati ponovno. Brez datoteke `.sln` v VS ne moremo zgraditi rešitve, medtem ko jo brez `.suo` datoteke lahko.



Slika 49: Odnos rešitve, njenih projektov in njihovih dokumentov

Na sliki 50 in sliki 51 vidimo rešitvi, ki vsebujeta projekte z določenimi datotekami. Te so se samodejno ustvarile ob kreiranju projekta. V nadaljevanju si bomo podrobneje pogledali tiste datoteke, ki se nam ustvarijo pri WPF in Windows Forms aplikacijah. Več o raziskovalcu rešitve najdete v [21] in [22].

Na sliki 50 si poglejmo enostavni primer, ko imamo rešitev, ki jo sestavlja en projekt z imenom `ImeProjektaWPF`. Ta vključuje več datotek. Te datoteke so razvrščene v projektu po kategorijah v mapah (`Properties`, `References`) in seznamu projektnih datotek (okna aplikacije – `MainWindow`, `App.xaml`, `App.config`).

Ko razvijamo aplikacijo in delamo na projektu, uporabljamo standardne gradnike (Form, Button in podobne). Če želimo uporabiti tudi gradnike iz drugih knjižnic (npr. `System.Windows.Forms`, `System.Data.SqlClient` …), moramo te vključiti v mapo `References` projekta. V nadaljevanju bomo videli, da se določene dodajo samodejno, ob kreiranju projektov.

Znotraj mape `Properties` imamo datoteko `AssembleyInfo.cs`, v kateri so vpisane različne informacije o aplikaciji (ime, opis, različica …). Vsi podatki znotraj te datoteke se ob določenih spremembah v projektu samodejno posodobijo, zato nam je večinoma ni treba urejati. Poleg te datoteke imamo v mapi `Properties` tudi datoteki `Resources.resx` in `Settings.settings`. V datoteki `Resources.resx` lahko shranimo podatke o lokaciji datotek, ki jih uporabljamo v projektu, kot so to npr. slike, zvočni posnetki, ikone. V datoteki `Settings.settings` urejamo nastavitve aplikacije. Te lahko urejamo kot nastavitve aplikacije, kar pomeni, da spremembe veljajo za vse uporabnike aplikacije, ali pa jih urejamo tako, da nastavitve veljajo samo za določene uporabnike.

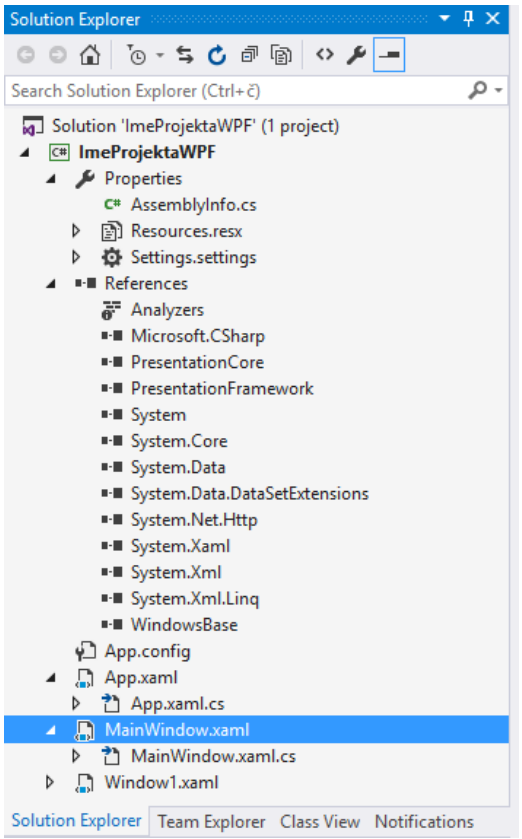
V mapi `References` so navedene knjižnice razredov, ki jih uporabljamo v aplikaciji. Kot smo že povedali, so običajno te knjižnice razredov iz ogrodja .NET. Občasno pa uporabimo tudi takšne, ki jih sami ustvarimo. Če ob ustvarjanju novega projekta izberemo okenske aplikacije, se nam v mapi samodejno dodajo tiste knjižnice, za katere okolje VS predvideva, da jih bomo potrebovali pri tej vrsti aplikacije. Pri WPF in Windows

Forms potrebujemo knjižnice, s katerimi lahko ustvarjamo GUV (gradniki, okna, grafično urejanje …), delo s podatki (podatkovne baze) … Poleg referenc, ki se samodejno dodajo ob izbiri vrste projekta, pa lahko v projekt vključimo še druge (glej razdelek Okenske aplikacije, kjer govorimo o predlogah).

Datoteka `App.config` je prisotna tako v WPF kot Windows Forms aplikacijah. V tej datoteki so zapisane nastavitve aplikacije, ki jih urejamo v datoteki `Settings.Settings`, kot je npr. povezava do podatkovne baze. Denimo da imamo aplikacijo, ki prikazuje podatke iz baze računalnika `x`. Poleg ene baze na računalniku x pa imamo ostale baze na računalniku `y`. Ker ne želimo imeti podatkovnih baz na dveh računalnikih, temveč želimo imeti vse podatkovne baze na enem računalniku, prenesemo podatkovno bazo iz računalnika `x` na `y`. Ko po prenosu baze znova zaženemo aplikacijo, ugotovimo, da le-ta ne prikazuje podatkov, saj je povezava do baze računalnika `x` prekinjena. Da rešimo nastalo težavo, lahko v tej datoteki spremenimo povezavo iz računalnika `x` na `y`.

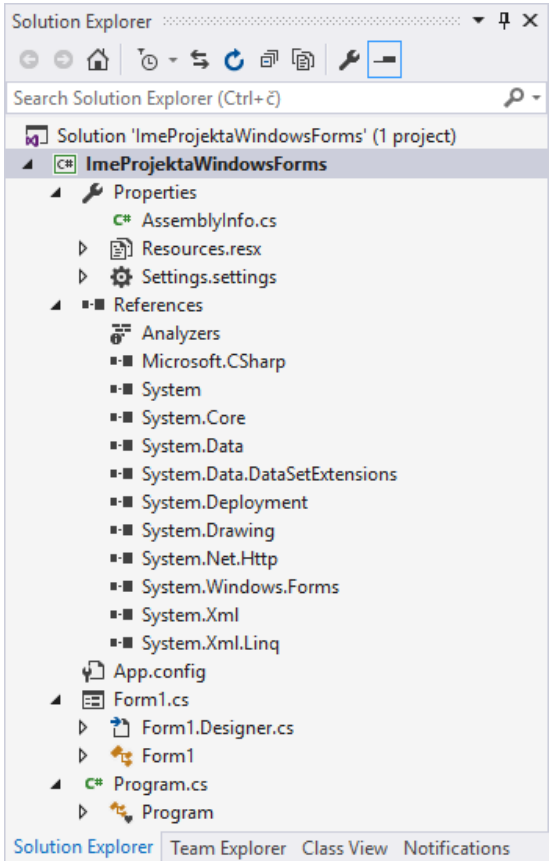
Datoteki `Program.cs` in `App.xaml` smo že podrobneje opisali v razdelku o okenskih aplikacijah, ko smo govorili o predlogah. Poleg teh dveh datotek pa se nam pri izbiri okenskih aplikacij samodejno ustvarita še datoteki `MainWindow.xaml` (WPF) in `Form1.cs` (Windows Forms). V teh dveh datotekah urejamo programski del aplikacije kot tudi videz s postavitvijo gradnikov na okna oz. obrazce.

Če primerjamo sliki 50 in 51 vidimo, da nam raziskovalec rešitev v obeh primerih prikazuje zelo podoben seznam datotek. Razlike se pojavijo v referencah do knjižnic razredov (`References`) in pri imenu ter vrsti datotek, kjer urejamo GUV.



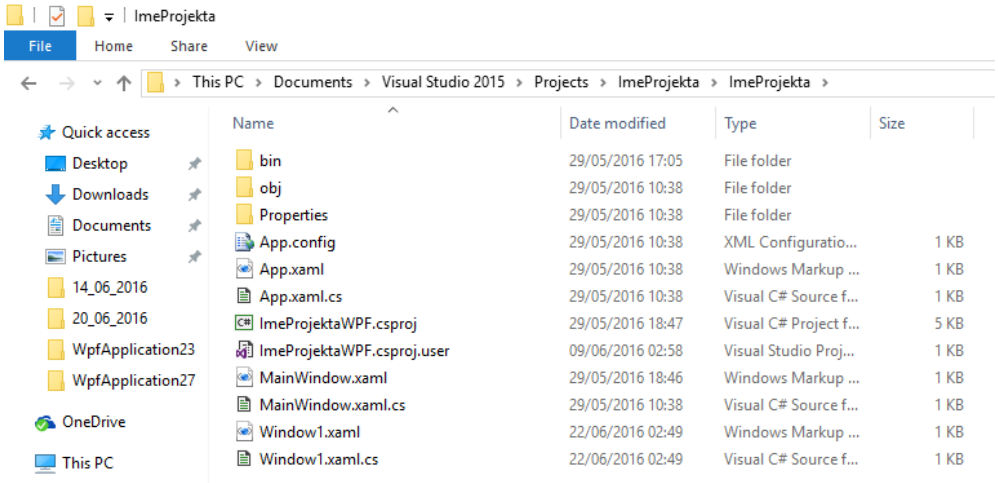
Slika 50: Datoteke, ki se samodejno ustvarijo v raziskovalcu rešitve WPF aplikacije

V raziskovalcu vidimo, da so datoteke, s katerimi urejamo uporabniški vmesnik, ločene od datotek, kjer zapišemo delovanje aplikacije. Tako imamo pri WPF aplikacijah datoteke, s katerimi urejamo grafične dele, označene s končnico `.xaml` in datoteke, kjer so opisana navodila aplikacije, s končnico `.xaml.cs`. Pri Windows Forms so te označene kot `.cs` (navodila aplikacije) in `.Designer.cs` (zapis grafičnega dela).



Slika 51: Datoteke, ki se samodejno ustvarijo v raziskovalcu rešitve Windows Forms aplikacije

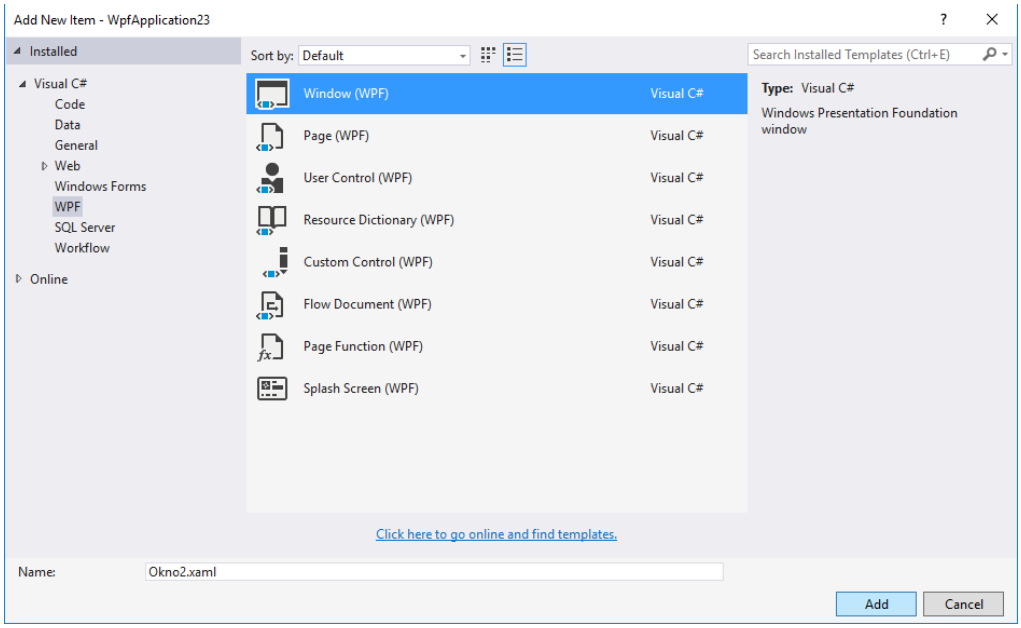
Razvojno okolje na računalniku ustvari mapo rešitve in znotraj nje mape s projekti, vključenimi v to rešitev. Če primerjamo sliki 50 in 52 vidimo, da so večinoma vse datoteke iz mape projekta na računalniku prisotne tudi v raziskovalcu rešitve.



Slika 52: Datoteke WPF projekta, shranjene na računalniku

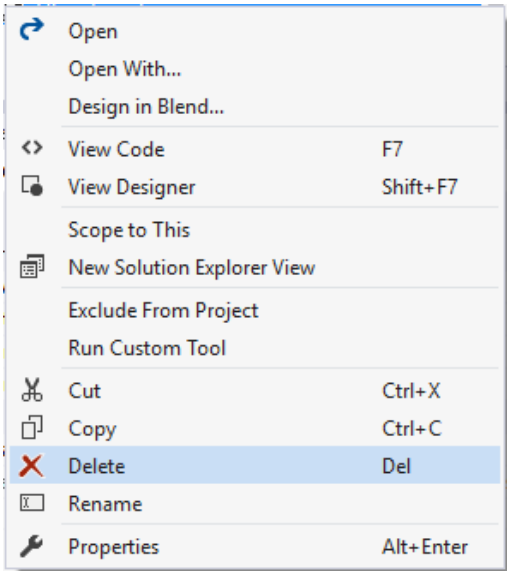
Ker imamo večinoma v GUV aplikacijah prisotno več kot eno okno (obrazec), moramo med razvojem v raziskovalcu dodati dodatna okna (obrazce). To storimo tako, da z desnim miškinim klikom na projekt izberemo možnost `Add`, kjer izbiramo med več možnostmi. Ker želimo v projekt dodati novo prazno okno (obrazec), izberemo možnost `New Item …` Nato izberemo `Window` oz. `Windows Forms`, ki ga praviloma v polju Name preimenujemo in kliknemo na gumb `Add` (glej sliko 53). Poleg tega lahko projektu dodamo tudi okna (obrazce), ki smo jih ustvarili že prej v kakšnem drugem projektu. To storimo tako, da namesto možno-

sti `New Item` … izberemo `Existing Item` …, in poiščemo ustrezno datoteko z opisom okna (obrazca) na računalniku ter kliknemo na gumb `Add`.



Slika 53: Dodajanje novega okna projektu

V nadaljevanju si poglejmo, kako pravilno odstranimo in preimenujemo določene datoteke, vključene v projekt. Čeprav običajno datoteko, ki je ne potrebujemo, vržemo v koš iz mape na računalniku, pa le-tega pri projektih ne počnemo. Datoteke je treba odstraniti iz razvojnega okolja tako, da se vse povezave med datotekami uredijo v projektni datoteki. Če pogledamo sliko 52, vidimo, da imamo v projekt vključeno okno `Window1`. Recimo, da smo ugotovili, da ga ne potrebujemo. Če bi datoteki `Window1.xaml` in `Window1.xaml.cs` odstranili kar neposredno iz diska računalnika, bi nam razvojno okolje javilo napako, da ne najde okna `Window1`. Poglejmo, kako datoteko iz projekta pravilno odstranimo. Postopek je zelo podoben kot pri dodajanju. Desno kliknemo na ustrezno datoteko v raziskovalcu rešitve in izberemo `Delete` (slika 54). S tem okolje izbriše tako ustrezno datoteko na disku računalnika, kot tudi ustrezno spremeni vse potrebne datoteke, v katerih je bil sklic na to okno.



Slika 54: Odstranitev datoteke iz projekta

Podobno ravnamo tudi, če želimo datoteko preimenovati. Spet moramo spremembo opraviti s pomočjo Raziskovalca rešitve.

S pomočjo raziskovalca na koncu projekt tudi pripravimo za objavo aplikacije za namestitev (več o tem v razdelku Priprava aplikacije za namestitev).

Gradniki

Z odlaganjem gradnikov iz okna orodjarne na obrazec (okno) ustvarimo grafično podobo programa. Pri tem si v VS okolju pomagamo z gradniki, ki jih imamo v oknu orodjarna (`Toolbox`). Le-to najdemo na levi strani okolja VS. Če je ne vidimo, jo lahko prikličemo s kombinacijo tipk `Ctrl + Alt + X`.

Pri odlaganju gradnikov na obrazce (okna) se samodejno ustvari koda, ki ustreza obrazcu (oknu) in gradnikom, ki jih odlagamo nanj. Pri Windows Forms aplikacijah imamo to kodo zapisano v jeziku C#, medtem ko je pri WPF aplikacijah ta koda zapisana v jeziku XAML.

Gradniki so vsi elementi, ki so prisotni na GUV, vidni ali ne. Velika večina gradnikov je na uporabniškem vmesniku vidnih, nekateri pa so vidni samo med razvojem projekta. Ti elementi so že pripravljeni sestavni deli programa in so namenjeni za določena opravila.

Izbor gradnikov, ki so nam na voljo, je odvisen od tipa aplikacije, ki jo razvijamo.

Pri WPF aplikacijah se grafična podoba aplikacije zapiše v datoteko s končnico `.xaml` (npr. `MainWindow.xaml`). Kot smo povedali že v razdelku o okenskih aplikacijah, lahko poleg načina povleci in odloži, gradnike dodajamo tudi z zapisom neposredno v datoteko XAML.

Pri Windows Forms aplikacijah se grafična podoba aplikacije ustvari v datoteki s končnico `.Designer.cs` (npr. `Form1.Designer.cs`). Če ga dodamo v datoteko ročno (kar ni priporočljivo), ga vidimo, ko preklopimo na način pogleda za oblikovanje `[Design]`.

Denimo da smo na grafični vmesnik dodali tri gradnike. Poglejmo, kakšen je zapis za Windows Forms in WPF aplikacije. Ti trije gradniki, ki jih bomo uporabili, so oznaka (`Label`), vnosno polje (`TextBox`) in gumb (`Button`). Na slikah 55 in 56 vidimo kodo, ki se samodejno ustvari ob dodajanju gradnikov na okno oz. obrazec.

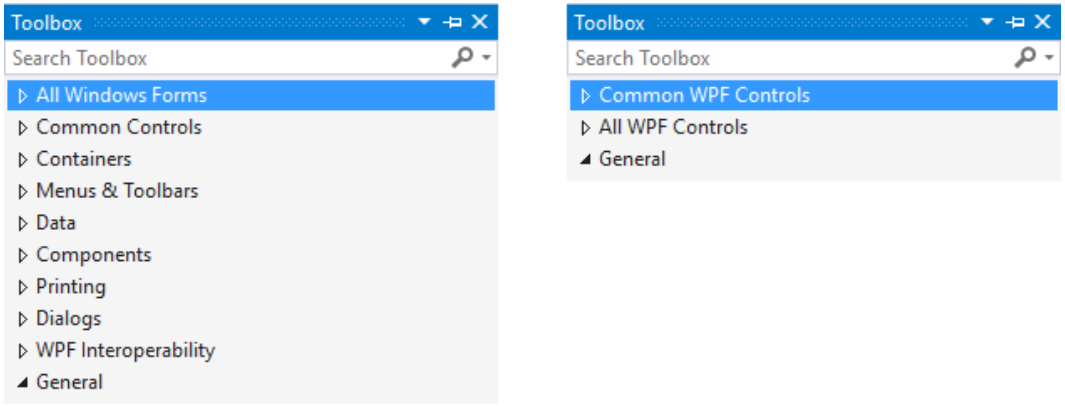
```
8         Title="MainWindow" Height="350" Width="525">
9     <Grid>
10        <Label x:Name="oznaka" Content="Oznaka" HorizontalAlignment="Left" Margin="60,49,0,0" VerticalAlignment="Top"/>
11        <TextBox x:Name="vnosnoPolje" HorizontalAlignment="Left" Height="23" Margin="60,80,0,0" TextWrapping="Wrap" Text="Vpiši ime" VerticalAlignm
12        <Button x:Name="gumb" Content="Moj gumb" HorizontalAlignment="Left" Margin="60,108,0,0" VerticalAlignment="Top" Width="75"/>
13
14    </Grid>
15 </Window>
```

Slika 55: Samodejno ustvarjen zapis odloženih gradnikov na oknu WPF aplikacije v jeziku XAML

```
31              this.besedilnoPolje = new System.Windows.Forms.TextBox();
32              this.oznaka = new System.Windows.Forms.Label();
33              this.gumb = new System.Windows.Forms.Button();
34              this.vnosnoPolje = new System.Windows.Forms.TextBox();
35              this.SuspendLayout();
36              //
37              // besedilnoPolje
38              //
39              this.besedilnoPolje.Anchor = ((System.Windows.Forms.AnchorStyles)((System.Windows.Forms.AnchorStyles.Top | System.Windows
40              this.besedilnoPolje.Location = new System.Drawing.Point(172, 60);
41              this.besedilnoPolje.Name = "besedilnoPolje";
42              this.besedilnoPolje.Size = new System.Drawing.Size(100, 20);
43              this.besedilnoPolje.TabIndex = 2;
44              //
45              // oznaka
46              //
47              this.oznaka.Location = new System.Drawing.Point(80, 20);
48              this.oznaka.Margin = new System.Windows.Forms.Padding(5);
49              this.oznaka.Name = "oznaka";
50              this.oznaka.Size = new System.Drawing.Size(192, 13);
51              this.oznaka.TabIndex = 1;
52              this.oznaka.Text = "Besedilo znotraj oznake se ne spremija.";
53              this.oznaka.MouseLeave += new System.EventHandler(this.label1_MouseLeave);
54              this.oznaka.MouseMove += new System.Windows.Forms.MouseEventHandler(this.label1_MouseMove);
55              //
56              // gumb
57              //
58              this.gumb.Anchor = ((System.Windows.Forms.AnchorStyles)((System.Windows.Forms.AnchorStyles.Top | System.Windows.Forms.An
59              this.gumb.Location = new System.Drawing.Point(197, 100);
60              this.gumb.Name = "gumb";
61              this.gumb.Size = new System.Drawing.Size(75, 23);
62              this.gumb.TabIndex = 0;
```

Slika 56: Del samodejnega zapisa odloženih gradnikov Windows Forms aplikacije v C# jeziku

Pri Windows Forms aplikacijah imamo gradnike v orodjarni razdeljene v več posameznih skupin glede na njihov namen (`All Windows Forms`, `Common Controls`, `Containers`, `Menus & Toolbars`, `Data` …). Pri WPF aplikacijah pa imamo gradnike razdeljene le v dve skupini (slika 57). S klikom na trikotnik pred imenom skupine se nam pokažejo gradniki, ki so vsebovani znotraj skupine. V njej nato poiščemo gradnik, ki ga želimo dodati aplikaciji. Ko izberemo npr. gradnik gumb (`Button`), ga označimo v orodjarni in kliknemo z miško še na mesto, kjer bi radi imeli gumb na obrazcu (oknu). V primeru, da želimo gradnik naknadno prestaviti, to naredimo tako, da ga označimo in nato z miško poberemo ter odložimo na novo mesto. Če ugotovimo, da na obrazcu (oknu) določenega gradnika ne potrebujemo več, ga lahko preprosto odstranimo iz obrazca (okna). Gradnik označimo in na tipkovnici pritisnemo `Delete`, ali pa na označen gradnik kliknemo z desno miškino tipko in iz okna, ki se nam pojavi, izberemo `Delete`.

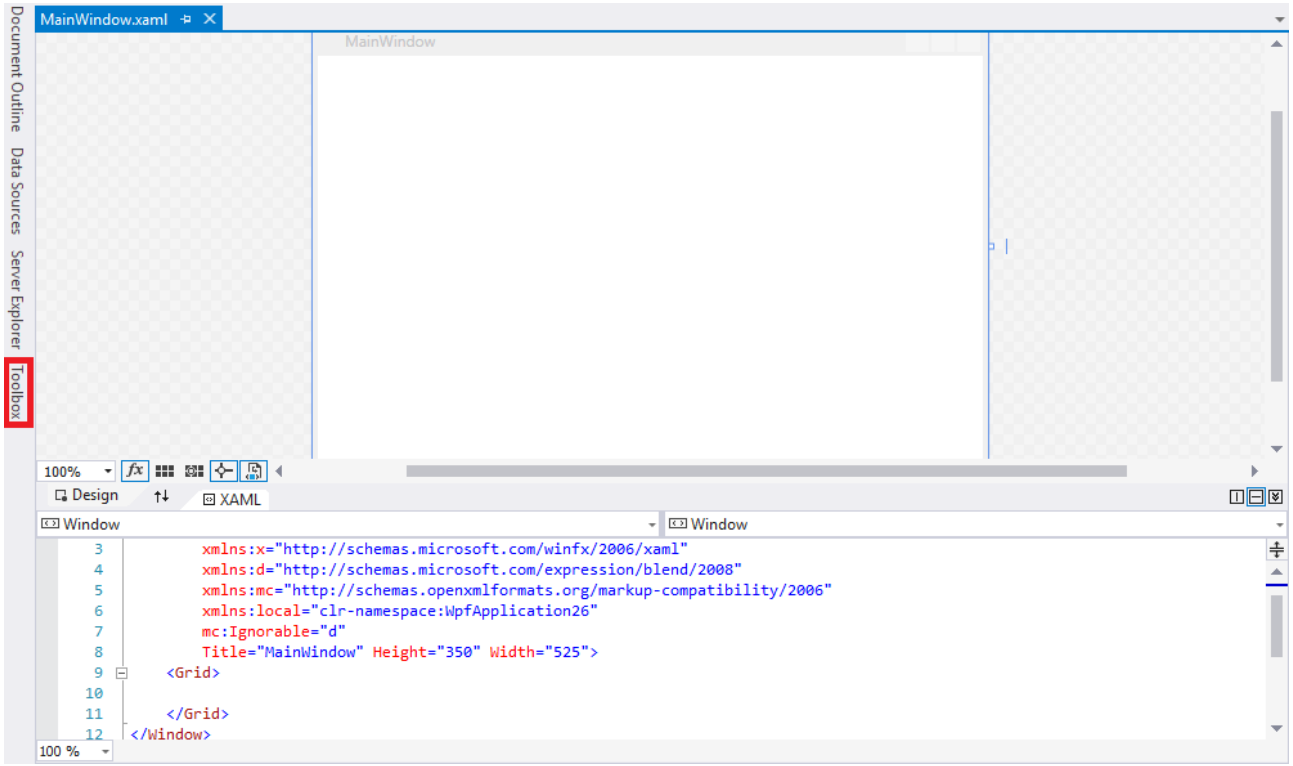


Slika 57: Skupine gradnikov glede na namen v orodjarni (levo Windows Forms, desno WPF)

V oknu lastnosti (`Properties`), ki je običajno desno spodaj, gradnikom nastavljamo različne začetne lastnosti. To okno je namenjeno prikazu in spreminjanju lastnosti in dogodkov gradnikov, odloženih na okno (obrazec). Lastnosti gradnikov v oknu Properties lahko razvrstimo na dva načina, po abecednem redu ali po kategorijah.

Vse lastnosti novo dodanih gradnikov na okno (obrazec) so privzete, ki jih nato običajno spremenimo. Ena izmed pomembnejših lastnosti gradnika je njegovo ime (`Name`). Ime gradnika je namreč oznaka, preko katere se sklicujemo na posamezne gradnike. VS gradnike samodejno poimenuje tako, da imenom na koncu doda zaporedno številko (npr. `label`, `label1`, `label2` …). Imena gradnikov lahko poljubno spremenimo in jim damo drugo, ustrezno ime. To je priporočljivo vsaj za gradnike, na katere se sklicujemo kasneje v kodi, saj lahko sicer izgubimo pregled nad gradniki.

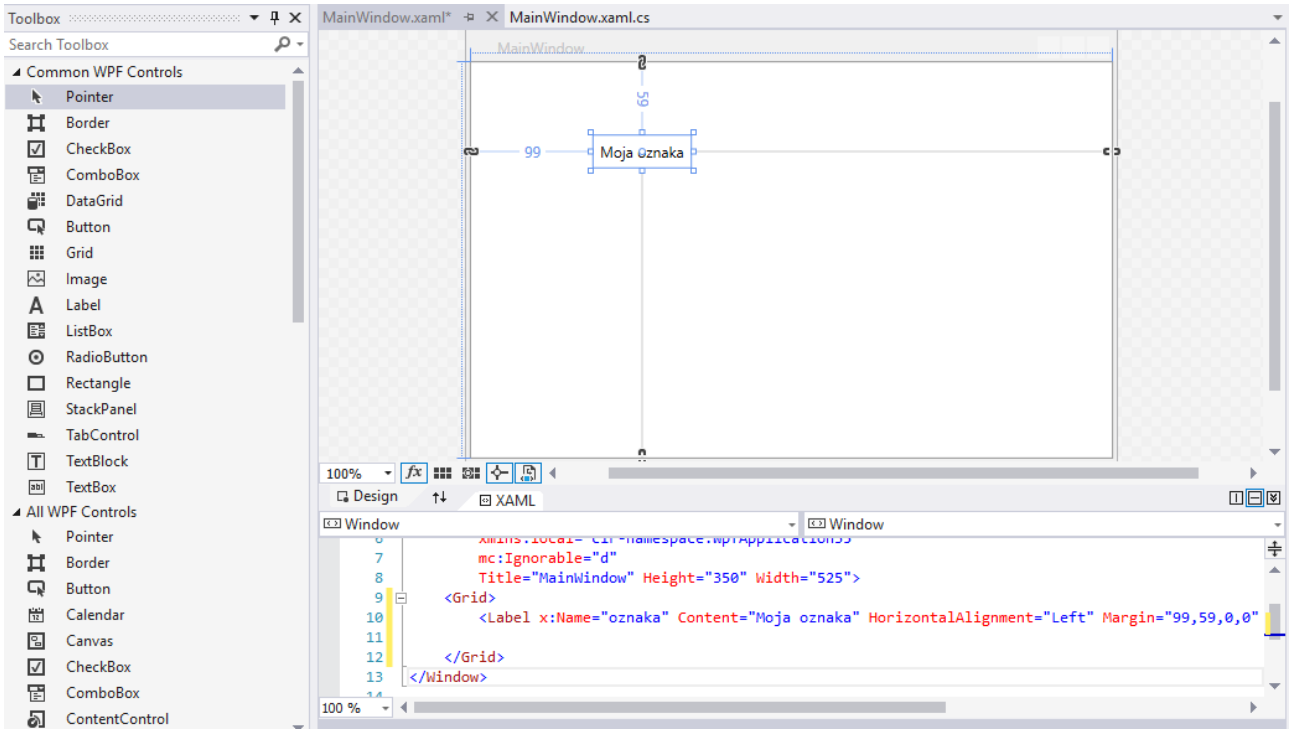
Spodaj si po korakih oglejmo, kako na okno WPF aplikacije dodamo že omenjene gradnike (oznaka, vnosno polje, gumb). Ko odpremo WPF projekt, imamo postavljen prvi gradnik, prazno okno. Na levi strani slike je z rdečo označen zavihek orodjarne.



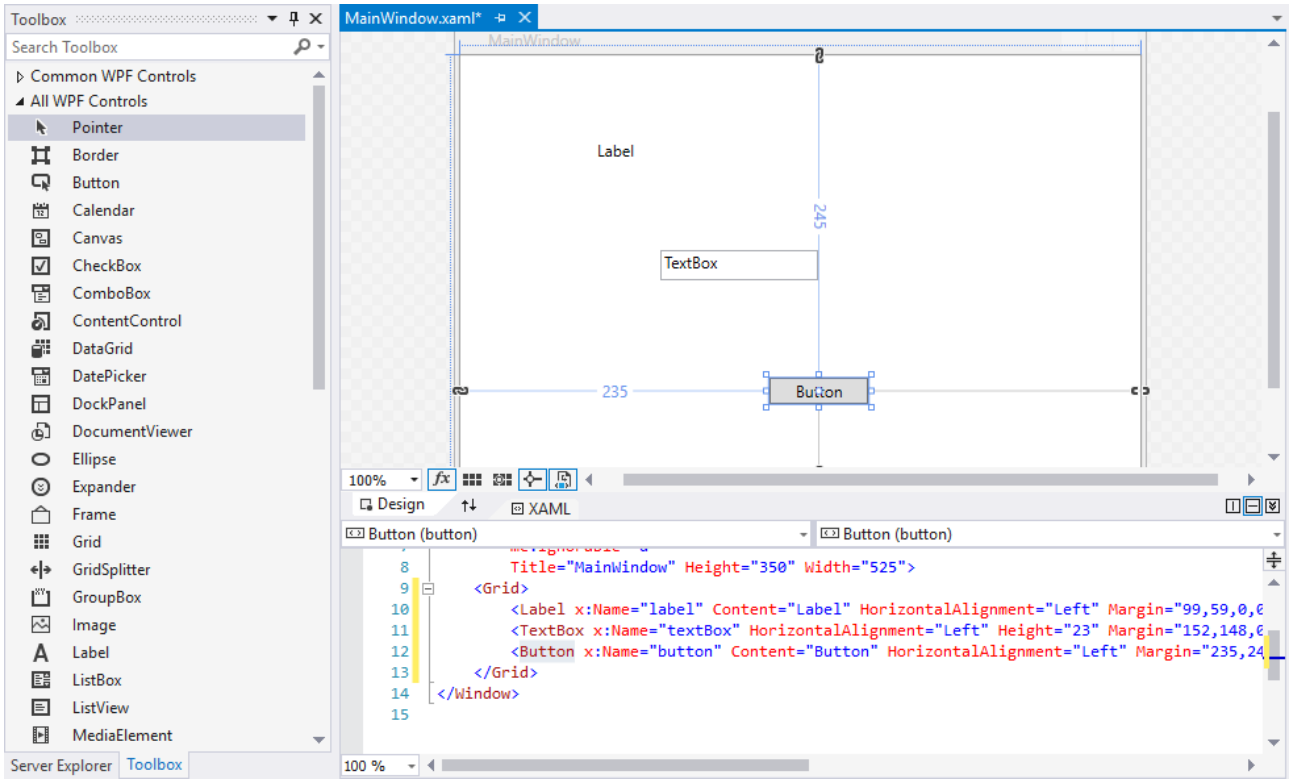
Slika 58: Prazno okno WPF aplikacije, na katerega dodamo gradnike iz orodjarne

Osnovno okno ima naslov `"MainWindow"`. Njegova vsebina je le značka `Grid`. To je značka, znotraj katere postavljamo gradnike. Takoj ko na okno prenesemo gradnik oznaka (`Label`), se v znački samodejno ustvari zapis (glej sliko 59).

Isti postopek ponovimo še za gradnika `TextBox` in `Button`. Na tak način postavimo na okno vse gradnike, za katere predvidevamo, da jih bomo potrebovali v aplikaciji. Ob tem se je samodejno ustvarila koda za vse 3 odložene gradnike. Vidimo (slika 60), da je v kodi zapisano ime posameznega gradnika (lastnost `x:Name`), kot tudi nekatere druge lastnosti.

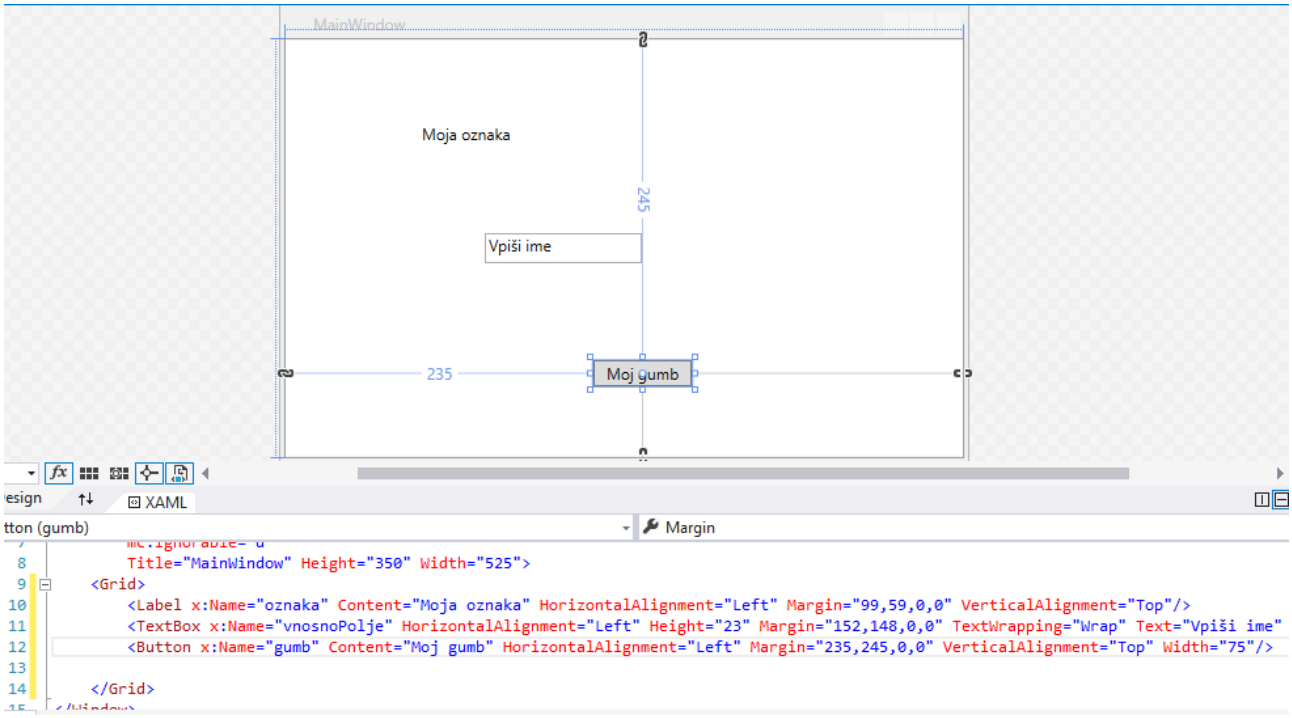


Slika 59: Dodani gradnik oznaka (Label) na okno WPF aplikacije



Slika 60: Trije različni gradniki, dodani na okno WPF aplikacije

Če želimo gumbu spremeniti napis, to storimo tako, da v XAML pri znački `Button` spremenimo vrednost lastnosti `Content`. Na sliki vidimo, da smo napis na gumbu s slike 60 spremenili v `"Moj gumb"`.



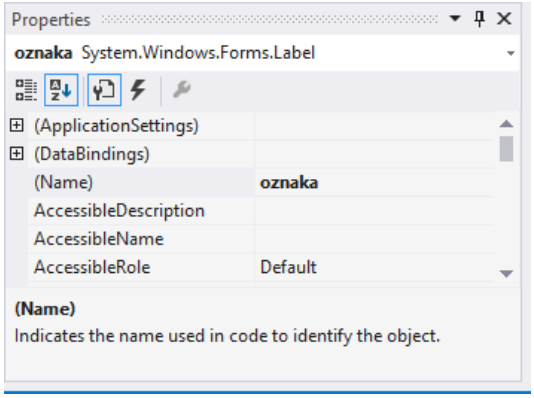
Slika 61: Sprememba napisa na gumbu

Lastnosti gradnikov pri WPF aplikacijah torej spreminjamo tako, da ustrezno spreminjamo kodo XAML.

Pri Windows Forms aplikacijah je zgodba nekoliko drugačna.

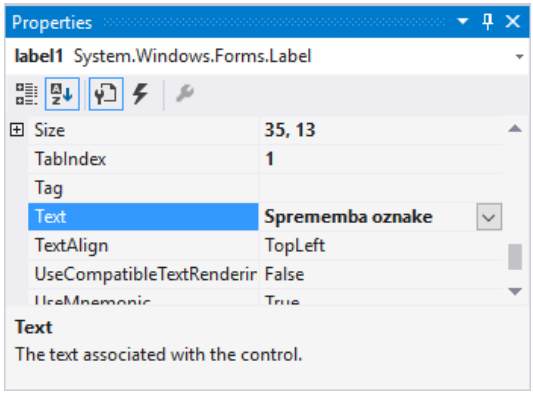
Če pogledamo sliko 62, vidimo, da pri Windows Forms aplikacijah urejamo gradnike preko okna `Properties`. Na sliki vidimo, da smo gradnik, ki mu je VS dal privzeto ime `label1`, preimenovali v `oznaka`.

V oknu `Properties` lahko vsakemu gradniku spremenimo lastnosti, ki jih ima na voljo. Ko kliknemo na posamezni gradnik, v desnem spodnjem kotu vidimo okno `Properties`, kjer spreminjamo lastnosti, ki jih ima gradnik. V oknu so vedno vidne le tiste lastnosti, ki jih za posamezno vrsto gradnika lahko nastavimo. Če smo sočasno označili več gradnikov, spremembe veljajo za vse označene gradnike. Seveda takrat določenih lastnosti (npr. Name) ne moremo spreminjati. Prav tako takrat, ko so izbrani gradniki različnih tipov, vidimo le tiste lastnosti, ki so skupne vsem vrstam označenih gradnikov.



Slika 62: Okno Properties, kjer nastavljamo lastnosti gradnikov

V oknu `Properties` po navadi nastavimo začetne nastavitve lastnosti (torej tiste, ki jih ima gradnik ob zagonu aplikacije). Vse te spremembe lastnosti, ki jih opravimo v oknu `Properties`, se dejansko zapišejo v obliki kode znotraj metode.



Slika 63: Sprememba lastnosti v oknu Properties

Če pogledamo sliko 63, kjer smo gradniku spremenili lastnost `Text`, vidimo, da je ta sprememba povzročila, da v metodi `InitializeComponent()`v datoteki `.Designer.cs` sedaj piše

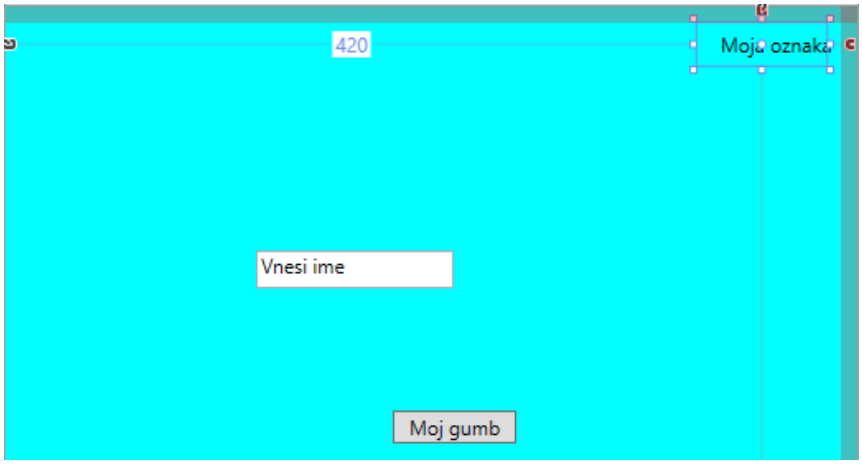
```
//
// oznaka
//
…
this.oznaka.Text = "Sprememba oznake";
```

Postavitev gradnikov

Pri gradnji uporabniškega vmesnika je zelo pomembno, kako gradnike razporedimo. Na sliki 64 vidimo, da so gradniki postavljeni praktično na poljubna mesta na oknu. Postavitev lahko urejamo ročno, ali pa si pomagamo z lastnostmi gradnikov in z določenimi gradniki, katerih osnovna naloga je, da vsebujejo druge gradnike (vsebniki). Eden od vsebnikov je že samo okno (obrazec). Pri WPF aplikacijah postavitev posameznih gradnikov znotraj vsebnika urejamo z lastnostmi `HorizontalAlignment`, `Margin`, `Padding` in `VerticalAlignment`.

Ročno položaj gradnika določimo tako, da ga izberemo in z miško povlečemo na ustrezno mesto. Na sliki 64 vidimo, da se zgornji in desni rob, ko jima približamo gradnik, obarvata. Ta obarvani del je na obeh robovih enako širok in nam pomaga, da gradnike razporedimo v enakomerni oddaljenosti od robov. Podobno se obarva tudi razmik med gradniki (slika 65). Na tak način lahko postavimo vse vidne gradnike. Slabost take postavitve je v tem, da se ob spremembi velikosti okna smiselnost postavitve poruši. Zaradi tega je priporočljivo gradnike postavljati v vsebnike (v WPF so to `StackPanel`, `WrapPanel`, `DockPanel` …), ki nato postavitev gradnikov prilagodijo velikosti okna.

Poglejmo si preprost primer ročne postavitve gradnikov WPF aplikacije. Recimo, da želimo imeti vse gradnike postavljene na desni strani okna, enega pod drugim. Slika 64 kaže, kako prestavljamo oznako. Podobno prestavimo še gradnika `TextBox` in `Button` tako, da bodo vsi trije gradniki en pod drugim.

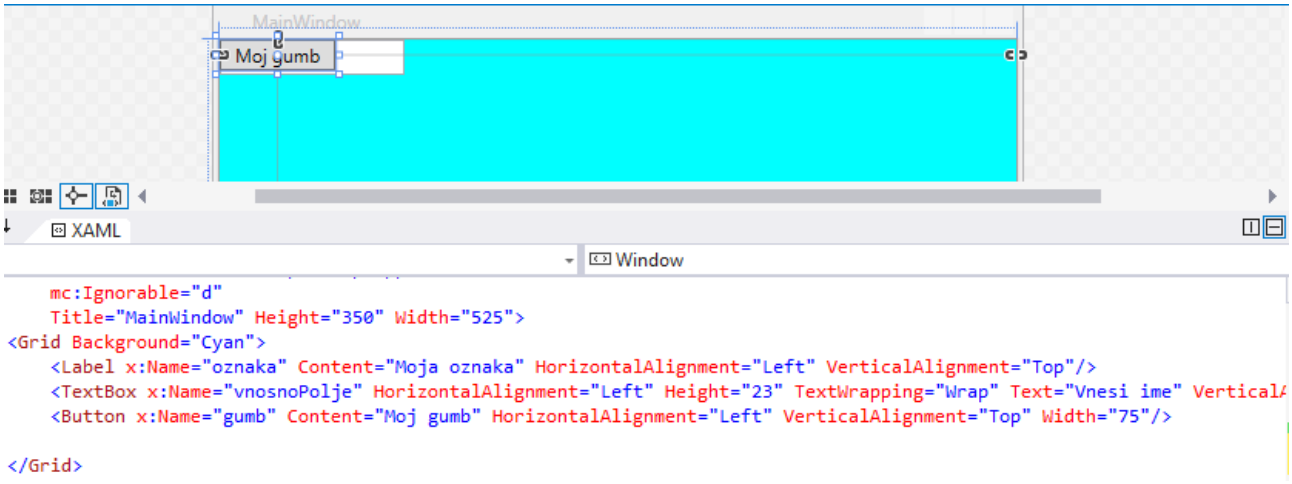


Slika 64: Ročna postavitev gradnikov na oknu WPF aplikacije

Slika 65: Postavitev gradnikov enega pod drugim

Če pogledamo XAML kodo, vidimo postavitev posameznih gradnikov še bolj podrobno. Vsi trije gradniki imajo privzete nastavitve za postavitev glede na nadrejen gradnik (vsebnik, ki jih vsebuje). To sta lastnosti `HorizontalAlignment` in `VerticalAlignment`. Vodoravna postavitev gradnika glede na nadrejen gradnik (`Grid`) je na levi strani (`HorizontalAlignment="Left"`), navpična postavitev pa zgoraj (`VerticalAlignment="Top"`). Ti dve lastnosti nam povesta, kje je izhodiščna točka posameznega gradnika. Zapis v lastnosti `Margin` pa nam pove oddaljenost od posameznih robov nadrejenega gradnika glede na njegovo izhodiščno točko.

```
<Grid>
    <Label x:Name="label" Content="Label" HorizontalAlignment="Left"
           Margin="469,10,0,0" VerticalAlignment="Top"/>
    <TextBox x:Name="textBox" HorizontalAlignment="Left" Height="23" Margin="387,41,0,0"
           TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top" Width="120"/>
    <Button x:Name="button" Content="Button" HorizontalAlignment="Left" Margin="432,69,0,0"
           VerticalAlignment="Top" Width="75"/>
</Grid>
```



Slika 66: Postavitev gradnikov

Težava pa se pojavi, ko npr. oknu na sliki 65, v izvajalnem načinu spremenimo velikost. Čeprav imamo na sliki 65 vse gradnike nastavljene, da ležijo ob desnem robu, pa so, ko okno povečamo, gradniki na sredini (glej sliko 67). To je zaradi tega, ker ko smo gradnik odlagali na okno, se je v XAML datoteki zapisala oddaljenost gradnika od levega zgornjega kota (postavitev gradnika znotraj okna lastnosti `HorizontalAlignment="Left"` in `VerticalAlignment="Top"` ter oddaljenost od te izhodiščne točke z lastnostjo `Margin`). Ko smo nato okno povečali, je ostala oddaljenost gradnika od posameznih robov nadrejenega gradnika še vedno ista in se ni prilagodila spremembi velikosti okna. Zato da se izognemo tem težavam, pri WPF aplikacijah uporabljamo vsebnike. Več o njih bomo povedali v razdelku o vsebnikih (`Containers`).

Slika 67: Postavitev gradnikov na oknu ob povečanju velikosti okna

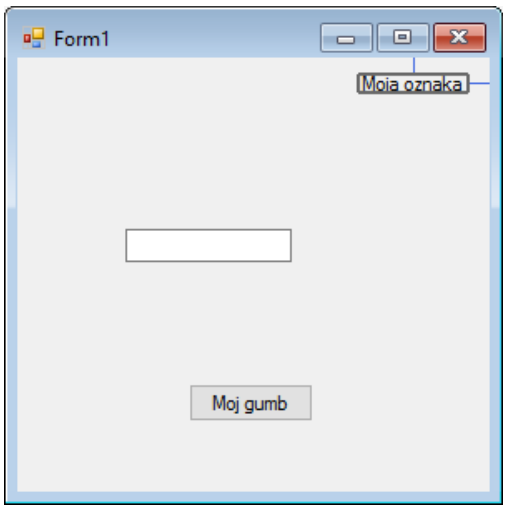
Če pa vsebnikov ne uporabljamo, moramo za naš primer spremeniti lastnost `HorizontalAlignment="Right"` in ustrezno spremeniti vrednost za `Margin` (slika 68). Tako se nam tudi, ko velikost okna povečamo, gradniki prikažejo tako, kot želimo, torej na desnem robu (glej sliko 69).

```
<Grid Background="Cyan">
    <Label x:Name="oznaka" Content="Moja oznaka" HorizontalAlignment="Right" Margin="20" VerticalAlignment="Top"/>
    <TextBox x:Name="vnosnoPolje" HorizontalAlignment="Right" Margin="20,60" Height="23" TextWrapping="Wrap" Text="Vnes
    <Button x:Name="gumb" Content="Moj gumb" HorizontalAlignment="Right" Margin="20,100" VerticalAlignment="Top" Width=

</Grid>
```

Slika 68: Sprememba lastnosti HorizontalAligment

Slika 69: Ne glede na spremembo velikosti okna so gradniki vedno ob desnem robu

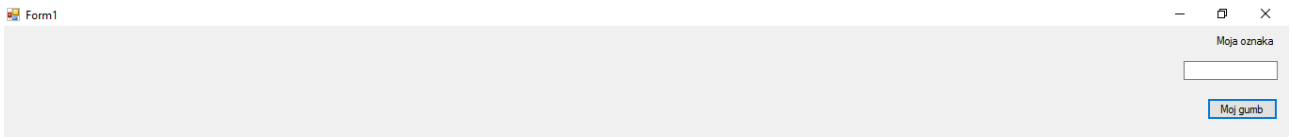
Sedaj si oglejmo še podoben primer, ko je uporabljen tip aplikacije Windows Forms. Tako kot pri WPF aplikaciji želimo imeti tudi tu vse gradnike postavljene ob desnem robu. Na sliki 70 vidimo, da postavljamo gradnike na obrazce podobno kot pri WPF aplikaciji, le oznake za oddaljenost so črte in ne odebeljeni robovi.



Slika 70: Ročna postavitev gradnikov na obrazcu Windows Forms aplikacije

Natančneje gradnik postavimo preko lastnosti `Location`. Lastnost `Location` je sestavljena iz dveh števil. Prva pomeni oddaljenost od levega roba vsebnika in druga oddaljenost od zgornjega roba vsebnika. Tako kot pri WPF aplikacijah se tudi tu v primeru, da obrazcu spremenimo velikost, spremeni grafična postavitev gradnikov na obrazcu. Da do tega ne pride, pa si pri Windows Forms aplikacijah pomagamo z lastnostma `Anchor` in `Dock`, ter vsebniki, kot so `TableLayoutPanel`, `SplitContainer`, `TabControl` …

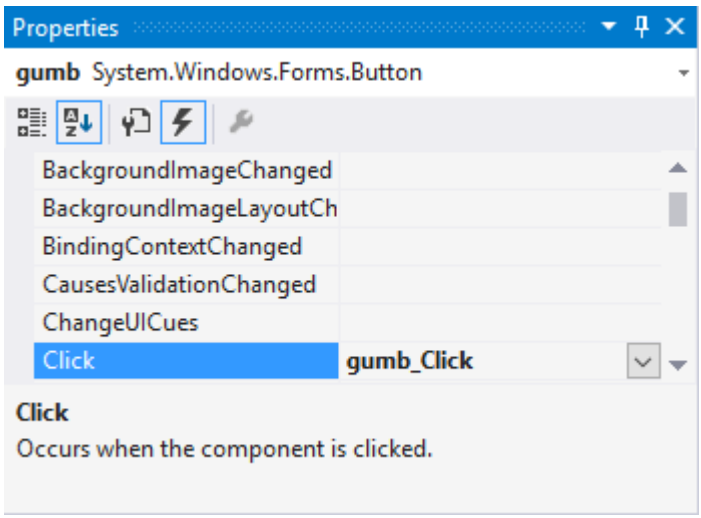
Lastnost `Anchor` je dejansko kombinacija lastnosti `HorizontalAlignment` in `VerticalAlignment`, le da je izražena kot par. Če torej v oknu `Properties` spremenimo lastnost `Anchor` iz privzete na `Top`, `Right`, so ob spremembi velikosti obrazca vsi prisotni gradniki postavljeni na desnem robu, podobno kot pri WPF aplikaciji (slika 71). Ob tem se spremeni tudi zapis znotraj datoteke s končnico `.Designer.cs`.



Slika 71: Postavitev gradnikov na desnem robu Windows Forms aplikacije

Določanje obnašanja gradnikov

Potem ko smo izdelali grafično podobo uporabniškega vmesnika, moramo nastaviti še obnašanje gradnikov. Kot smo povedali že v razdelku o dogodkovnem programiranju, so pri aplikacijah z GUV zelo pomembni dogodki. Gradnikom določimo, kako se morajo ob določenih dogodkih obnašati, torej kakšna programska koda se izvede ob tem dogodku. Te dogodke najdemo v oknu `Properties` (glej sliko 72). Ob kliku na ime dogodka se v spodnjem delu prikaže opis tega dogodka. Privzeto gradniki še ne reagirajo na nobenega od naštetih dogodkov. Če pa hočemo, da se ob določenem dogodku izvede neka koda, v seznamu dogodkov poiščemo ustrezen dogodek in z miško kliknemo nanj. V polju zraven imena dogodka nato zapišemo ime metode, ki se bo izvedla ob tem dogodku (npr. `gumbOK_potrdi`) in pritisnemo tipko `Enter`. Če pa uporabimo dvojni levi klik v vnosnem polju dogodka, pa nam VS ustvari metodo z imenom, ki je sestavljeno iz kombinacije imena gradnika in imena dogodka (npr. `gumb_Click`).



Slika 72: Samodejno ustvarjeno ime metode, ki se izvede ob kliku na gumb

Nato se nam pojavi koda v urejevalniku z ustvarjeno metodo, v katero moramo zapisati postopek, ki se izvede ob dogodku (slika 73).

```
private void gumb_Click(object sender, EventArgs e)
{

}
```

Slika 73: Prazna metoda, v katero vpišemo odziv na dogodek

Na sliki vidimo, da smo nastavili, da se ob kliku na gumb prikaže obvestilo, ki pove, da je uporabnik pritisnil na gumb.

```
private void gumb_Click(object sender, EventArgs e)
{
    MessageBox.Show("Pritisnili ste na gumb.");
}
```

Slika 74: Določeni postopek, ki se izvede ob kliku na gradnik gumb

Pri določenih gradnikih običajno nastavimo odziv na neki osnovni dogodek. Tako npr. praktično za vsak gumb povemo, kaj se zgodi, ko nanj kliknemo, za vnosno polje ustvarimo odziv ob spremembi zapisa v njem … Da je delo hitrejše, VS omogoča, da na gradnik le dvakrat kliknemo. S tem se avtomatično odpre urejevalnik za vpis kode za privzeti dogodek gradnika. Tako se nam npr. pri gradniku gumb že z dvojnim klikom nanj samodejno ustvari metoda `gumb_Click` in nam dogodka ni treba poiskati v seznamu dogodkov.

VS takrat, ko se ustvari metoda, ki pomeni odziv na določeni dogodek, poskrbi še za vrsto zadev. Tako npr. ob kliku na gumb `gumb` v datoteko `.Designer.cs` samodejno doda spodnjo kodo, s katero v izvajalnem okolju program natančno ve, katero metodo mora sprožiti in določi, da se ob kliku na specifični gumb izvede pravilni dogodek.

```
//
// gumb
//
…
this.gumb.Click += new System.EventHandler(this.gumb_Click);
```

V nadaljevanju si bomo na primerih pogledali nekaj gradnikov in razložili njihovo uporabnost. Najprej si bomo pogledali gradnike Windows Forms aplikacij.

Windows Forms

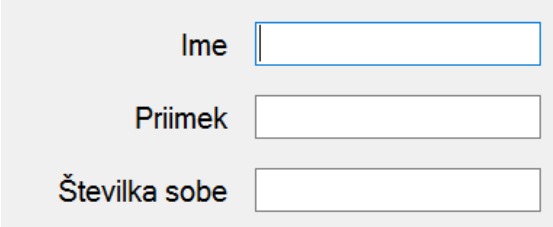
Kot smo že prej povedali, imamo v Windows Forms gradnike razdeljene na različne skupine glede na njihov namen. Zato si bomo pogledali nekaj najbolj uporabnih gradnikov posameznih skupin.

Najpogosteje uporabljeni gradniki: Common Controls

Znotraj te skupine imamo gradnike, ki so najbolj pogosto uporabljeni v Windows Forms aplikacijah. Ogledali si jih bomo le nekaj.

Label/Oznaka

Je eden izmed najbolj osnovnih in najbolj pogosto uporabljanih gradnikov. Uporabljamo ga za prikaz besedil znotraj oken (gradnikov tipa `Form`). Njegova osnovna lastnost je `Text`, s katero določimo, kakšen je napis na tej oznaki. Za več informacij o tem gradniku si poglejte [23].



Slika 75: Primer uporabniškega vmesnika z več oznakami in vnosnimi polji

Na sliki 75 je prikazano okno, kjer smo poleg treh vnosnih polj (več o tem tipu gradnikov v nadaljevanju) uporabili tri gradnike tipa `Label`. Z napisi na oznakah želimo uporabniku povedati, katere podatke naj vpiše v posamezna vnosna polja.

Pri gradniku `Label` sta poleg dogodka `Click` še najbolj uporabna dogodka `MouseMove` in `MouseLeave`. Z njuno pomočjo lahko na primer spremenimo besedilo na oznaki, ko nad oznako premaknemo miškin kazalec. Prvi dogodek se izvede, ko se miškin kazalec premakne nad pravokotno območje, ki ga zaseda oznaka, drugi pa, ko miškin kazalec zapusti območje oznake.

Tako lahko npr. ustvarimo kviz, kjer uporabnik s premikom miške nad oznako izve pravilni odgovor. Ko pa uporabnik umakne miško z zapisa, se zopet pojavi vprašanje. Na oznaki sprva piše besedilo "`Katerega leta se je rodil Nikola Tesla?`". Ko se z miško premaknemo nad oznako, pa se pokaže odgovor "`Nikola Tesla se je rodil leta 1856.`", ki pa se znova zamenja v vprašanje, ko z miško zapustimo pravokotno območje, ki ga zaseda oznaka.

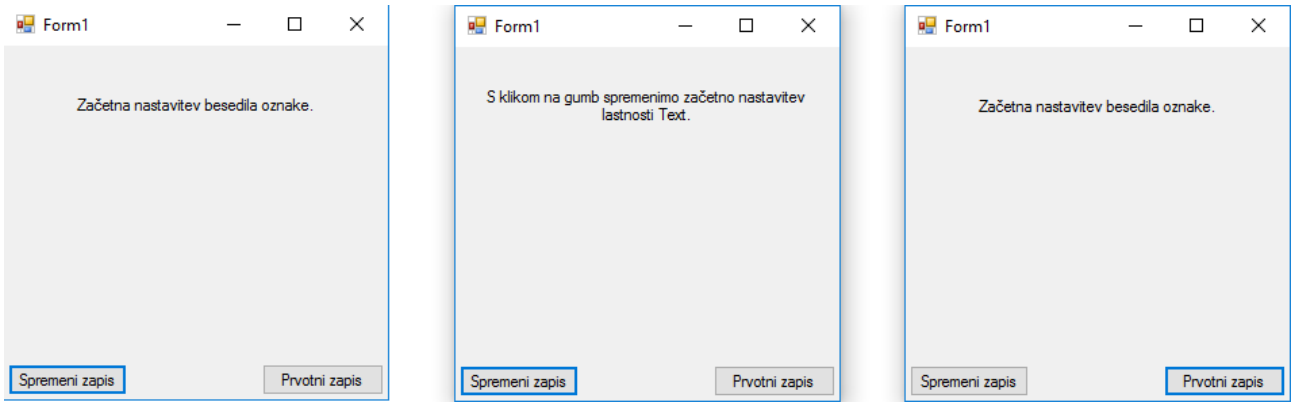
```
public Form1()
{
    InitializeComponent();
    oznaka.Text = "Katerega leta se je rodil Nikola Tesla?";
}

private void oznaka_MouseMove(object sender, MouseEventArgs e)
{
    oznaka.Text = "Nikola Tesla se je rodil leta 1856.";
}

private void oznaka_MouseLeave(object sender, EventArgs e)
{
    oznaka.Text = "Katerega leta se je rodil Nikola Tesla?";
}
```

Button/Gumb

V osnovi je ta gradnik prikazan kot pravokotnik z besedilom ali sliko znotraj njega. Večinoma v aplikacijah potrebujemo gumb zato, da potrdimo neki korak. Za več informacij o gradniku poglejte v [24].

Poleg lastnosti `Text`, s katero nastavimo, kaj na gumbu piše, je uporabna tudi lastnost `Image`, kjer na gumb dodamo sliko in lastnost `Background`, kjer določimo ozadje gumba.

Privzeti dogodek pri gumbu je dogodek `Click`. Ta se sproži, ko na gumb kliknemo z miško. Če pa je gumb že označen, lahko to storimo tudi s tipko `Enter` ali s tipko presledek.



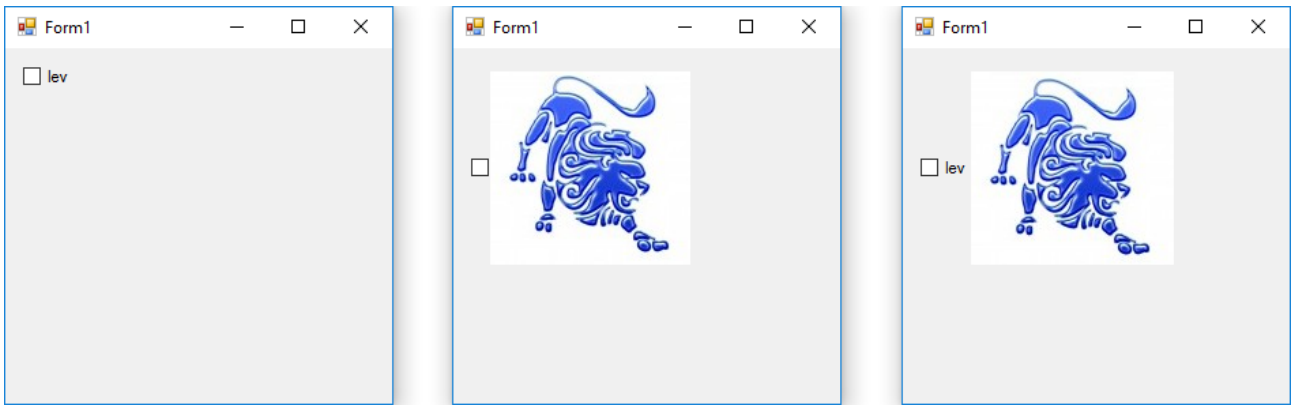
Slika 76: S klikom na gumb spremenimo zapis znotraj oznake

```
public Form1()
{
    InitializeComponent();
    gumbSpremeni.Text = "Spremeni zapis";
    gumbPovrni.Text = "Prvotni zapis";
}

private void gumb_Click(object sender, EventArgs e)
{
    oznaka.Text = "S klikom na gumb spremenimo začetno nastavitev lastnosti Text.";
}

private void gumbPovrni_Click(object sender, EventArgs e)
{
    oznaka.Text = "Začetna nastavitev besedila oznake.";
}
```

Na sliki 76 vidimo, da se besedilo oznake spreminja ob kliku na posamezen gumb, tako se ob kliku na gumb z napisom `"Spremeni zapis"` spremeni prvotno besedilo oznake v `"S klikom na gumb spremenimo začetno nastavitev lastnosti Text."`. Ko uporabnik klikne na gumb z napisom `"Prvotni zapis"`, pa se besedilo znotraj oznake spremeni v `"Začetna nastavitev besedila oznake."`.

CheckBox /Potrditveno polje

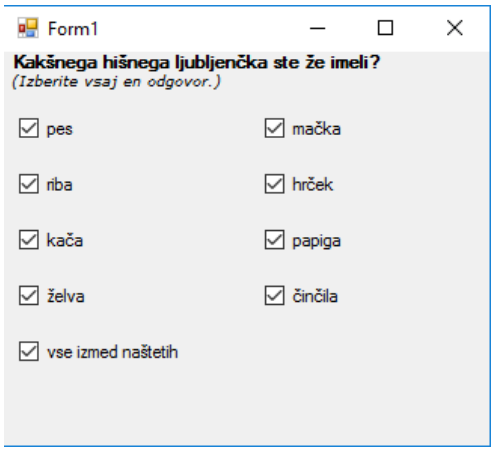
S tem gradnikom ponudimo uporabniku možnost, da izbere določene možnosti. V osnovi je ta gradnik zelo podoben gradniku `RadioButton`, ki ga bomo spoznali v nadaljevanju. Informacijo, ki jo nosi gradnik, predstavimo kot besedilo, sliko ali njuno kombinacijo (slika 77). Če želimo v obrazcu gradnik predstaviti samo besedilo, urejamo to v lastnosti `Text`. V primeru, da želimo namesto besedila gradnik predstaviti s sliko, moramo v lastnosti `Text` besedilo izbrisati in v lastnosti `Image` uvoziti sliko, s katero želimo predstaviti gradnik. Za kombinacijo slike in besedila pa v lastnosti `Text` zapišemo besedilo, ki bo predstavljeno, in uvozimo sliko v lastnosti `Image`. Pomembna je tudi lastnost `Checked`, ki pove, ali je gradnik označen ali ne.



Slika 77: Prikaz gradnika na obrazcu

Eden izmed najbolj uporabljenih dogodkov tega gradnika je `CheckedChanged`. Ta dogodek se izvede, ko se vrednost lastnosti `Checked` spremeni. Več informacij o gradniku najdete v [25].

Na sliki 78 vidimo, da uporabnika povprašamo, ali je že imel katerega od naštetih hišnih ljubljenčkov in mu ponudimo nekaj možnosti. Ena izmed naštetih možnosti je tudi `"vse izmed naštetih"`. Želimo, da takrat, ko uporabnik izbere (potrdi to polje), vsi preostali gradniki tudi postanejo potrjeni.



Slika 78: Vsi gradniki CheckBox označeni

Sedaj si poglejmo kodo. V njej vidimo, da na začetku uredimo slog in besedilo oznak. Nato vsem potrditvenim poljem nastavimo besedila. Pri metodi `ppVse_CheckedChanged` preverimo, ali je gradnik `ppVse` postal označen (`ppVse.Checked == true`). Če je, spremenimo lastnost `Checked` vsem preostalim gradnikom na oknu.

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

    //uredimo slog oznak in njun zapis
    FontFamily imePisave = new FontFamily("Microsoft Sans Serif");
    Font stil = new Font(imePisave, 8, FontStyle.Bold);
    oznaka.Font = stil;
    oznaka.Text = "Kakšnega hišnega ljubljenčka ste že imeli?";
    FontFamily imePisave2 = new FontFamily("Verdana");
    Font stil2 = new Font(imePisave2, 7, FontStyle.Italic);
    dodatnaOznaka.Font = stil2;
    dodatnaOznaka.Text = "(Izberite vsaj en odgovor.)";

    //potrditvenim poljem nastavimo besedila
    ppPes.Text = "pes";
    ppMacka.Text = "mačka";
    ppRiba.Text = "riba";
    ppHrcek.Text = "hrček";
```

```
    ppKaca.Text = "kača";
    ppPapiga.Text = "papiga";
    ppZelva.Text = "želva";
    ppCincila.Text = "činčila";
    ppVse.Text = "vse izmed naštetih";
}

private void ppVse_CheckedChanged(object sender, EventArgs e)
{
    //preverimo če je označeno potrditveno polje vse
    if (ppVse.Checked)
    {
        //v primeru da je, označimo vsa potrditvena polja
        ppPes.Checked = true;
        ppMacka.Checked = true;
        ppRiba.Checked = true;
        ppHrcek.Checked = true;
        ppKaca.Checked = true;
        ppPapiga.Checked = true;
        ppZelva.Checked = true;
        ppCincila.Checked = true;
    }
}
```

RadioButton /Radijski gumbi

V osnovi je gradnik zelo podoben gradniku `CheckBox`. Razlika med njima je le v tem, da lahko tu uporabnik izmed vseh gradnikov `RadioButton` označi izključno enega, ki je vsebovan znotraj ene skupine (vsebnika).

Kot pri drugih podobnih gradnikih, kjer je prisotno besedilo, urejamo prikazano besedilo v lastnosti `Text`. Slog in velikost pisave pa urejamo preko lastnosti `Font`. Izbirno polje je krog. Ta je običajno na levi strani, ob njem pa je pojasnilo. Tako kot pri gradniku `CheckBox` je pojasnilo lahko kombinacija besedila in slike. Če želimo položaj kroga spremeniti, to storimo v lastnosti `RightToLeft`.

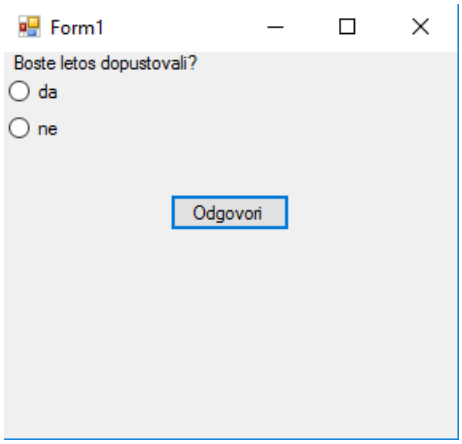
Tako kot pri `CheckBox` nam tudi tu lastnost `Checked` pove, ali je gradnik tipa `RadioButton` označen (izbran) ali ne. Vendar takrat, ko izberemo drugega, se prej izbrani odznači. Sočasno je torej v skupini izbran le eden.

Najpogosteje pri gradniku `RadioButton` uporabljamo dogodek `CheckedChanged`, kjer opišemo postopek, ki se zgodi v primeru, ko se spremeni stanje lastnosti `Checked`. Več informacij o gradniku dobite v [26].

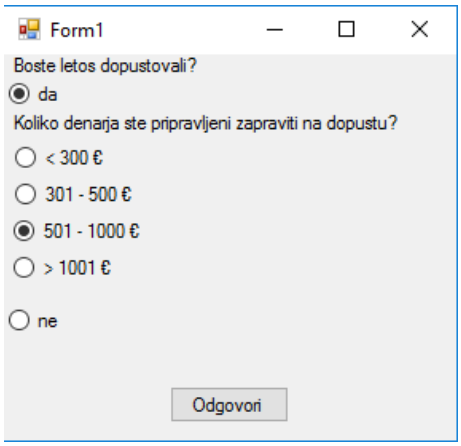
Poglejmo si primer aplikacije, kjer uporabimo gradnik `RadioButton` (slika 79).

Želimo sestaviti vprašalnik, kjer bomo uporabnike spraševali, ali nameravajo dopustovati. Le v primeru, če bodo odgovorili z `da`, jim bomo ponudili dodatno izbiro, kjer bodo označili znesek, ki ga bodo namenili za dopust.

Potrebovali bomo torej dve skupini radijskih gumbov. V prvi bosta le dva gumba (`da`/`ne`), v drugi pa zneski (slika 80). Slednja skupina bo na začetku skrita in bo postala vidna le, če uporabnik označi iz prve skupine radijskih gumbov odgovor `da`. Prva skupina je kar okno samo, drugo skupino pa bomo ustvarili v pregledni plošči `zneskiZaDopust` (več o tipu gradnika `Panel` v razdelku Vsebniki).



Slika 79: Aplikacija ob zagonu, ko od uporabnika pričakujemo, da označi eno izmed možnosti



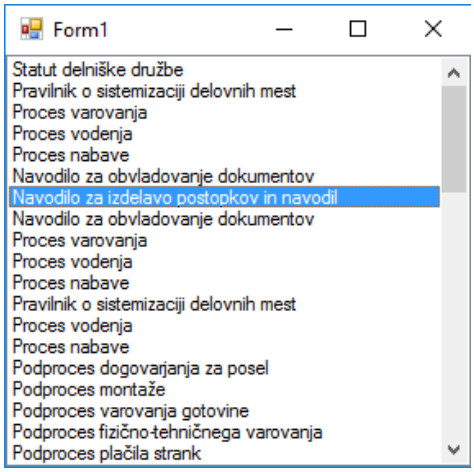
Slika 80: Izbrana gradnika

Iz spodnje kode vidimo, da smo uporabili dogodek `CheckedChanged`, kjer v primeru, da uporabnik izbere `radioButtonDa`, postaneta oznaka `oVprasanjeDa` in plošča `zneskiZaDopust` vidni. V primeru, da uporabnik zopet izbere gradnik `radioButtonNe`, se gradniki, ki so prikazani ob izboru `radioButtonDa`, zopet skrijejo. Čeprav smo napisali, da lahko uporabnik izbere samo eno možnost, pa vidimo na sliki 80, da sta izbrana dva gradnika. To je zaradi tega, ker so nekateri gradniki `RadioButton`, vsebovani v nadrejenem gradniku `Panel` (`zneskiZaDoupst`), nekateri pa v gradniku Form. V gradniku `zneskiZaDopust` imamo nato postavljene 4 gradnike `RadioButton`, izmed katerih lahko izberemo samo enega.

```
private void radioButtonDa_CheckedChanged(object sender, EventArgs e)
    {
        if (radioButtonDa.Checked == true)
        {
            oVprasanjeDa.Visible = true;
            zneskiZaDopust.Visible = true;
        }
        else
        {
            oVprasanjeDa.Visible = false;
            zneskiZaDopust.Visible = false;

        }
    }
```

ListBox /Izbirni seznam

Gradnik uporabljamo v aplikacijah, ko želimo uporabniku omogočiti pregled in izbiro podatkov iz nekega seznama.



Slika 81: Lastnost SelectionMode, nastavljena na One

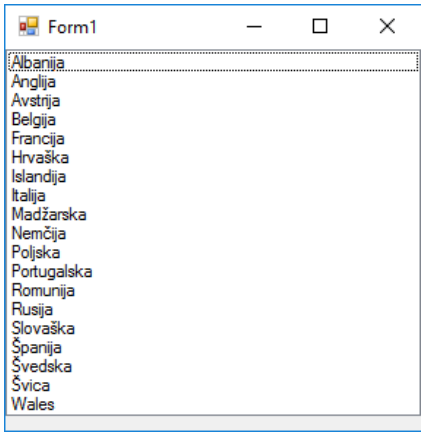
Osnovna lastnost tega gradnika je `Items`. To je seznam, ki vsebuje ustrezne podatke. Seznam lahko spreminjamo tudi programsko, s pomočjo metode `Add`.

Gradniku lahko nastavimo, da ima uporabnik možnost iz seznama izbrati eno ali več postavk. To urejamo v lastnosti `SelectionMode`, kjer je privzeta vrednost `One`. Takrat lahko uporabnik izbere samo en podatek iz seznama (slika 81). Če tej lastnosti spremenimo vrednost na `MultiSimple`, pa lahko uporabnik izbere več podatkov, tako da vsakega označi s posameznim klikom. V primeru, da to lastnost nastavimo na `MultipleExtended`, pa lahko uporabnik izbiro opravi na znani način s klikanjem in s pomočjo tipk `Ctrl` in `Shift`. Če pa želimo (začasno) preprečiti izbiro kateregakoli podatka, nastavimo lastnost `SelectionMode` na `None`.

Iz gradnika lahko tudi dobimo podatke, ki jih označimo v seznamu. Pri tem si pomagamo z lastnostjo `SelectedItems`. Poglejmo si primer. V gradnik smo vstavili nekaj držav, ki smo jih letos lahko spremljali na evropskem prvenstvu v nogometu. Z metodo `Add` podatke dodajamo na konec. Ker želimo, da so podatki v seznamu urejeni, jih uredimo tako, da lastnost `Sorted` nastavimo na `True` (slika 82).

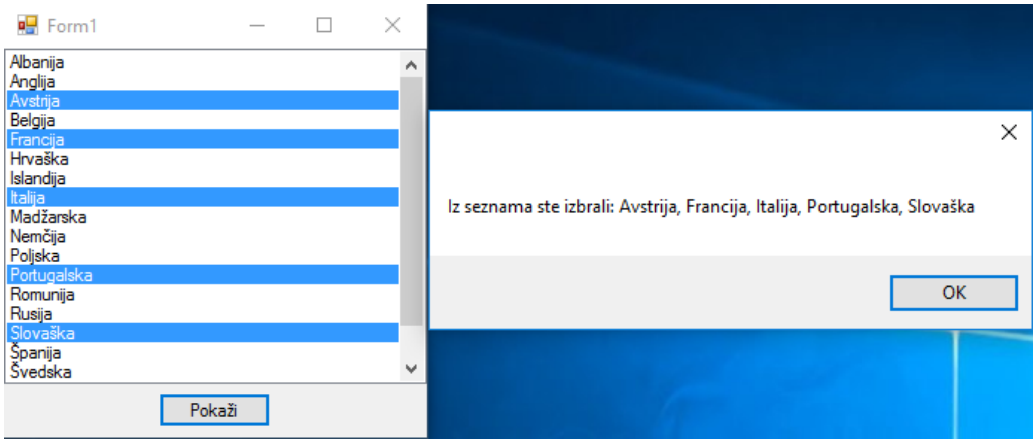
```
isEuro.Items.Add("Italija");
isEuro.Items.Add("Hrvaška");
isEuro.Items.Add("Madžarska");
isEuro.Items.Add("Anglija");
isEuro.Items.Add("Belgija");
isEuro.Items.Add("Wales");
isEuro.Items.Add("Albanija");
isEuro.Items.Add("Francija");
isEuro.Items.Add("Romunija");
isEuro.Items.Add("Švica");
isEuro.Items.Add("Rusija");
isEuro.Items.Add("Slovaška");
isEuro.Items.Add("Nemčija");
isEuro.Items.Add("Poljska");
isEuro.Items.Add("Islandija");
isEuro.Items.Add("Švedska");
isEuro.Items.Add("Španija");
isEuro.Items.Add("Portugalska");
isEuro.Items.Add("Avstrija");
```

```
listBox1.Sorted = true;
```



Slika 82: Države, razvrščene po abecedi

Poglejmo si še en primer, kjer označimo elemente iz seznama in s klikom na gumb prikažemo sporočilo, katere države iz seznama smo izbrali.



Slika 83: S klikom na gumb v oknu zapišemo označene države iz seznama

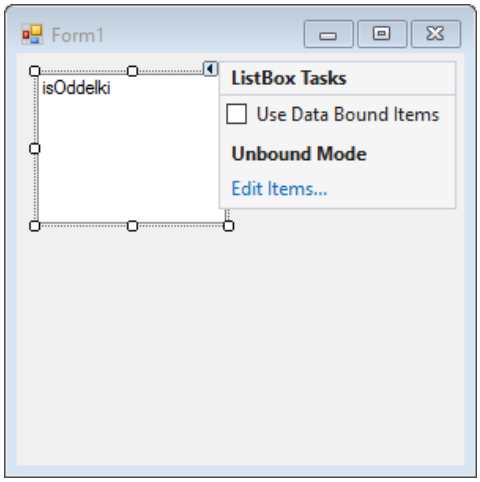
V kodi vidimo, da smo ustvarili metodo `gPokazi_Click()`, ki nam ob kliku gumba vrne okno, v katerem imamo izpisano obvestilo, katere države so označene na seznamu.

Lastnost `SelectedItems` nam vrne zbirko označenih elementov. Po njej lahko iteriramo z zanko `foreach`. Na ta način vse države, ki so označene v izbirnem seznamu, zapišemo v spremenljivko `izpis`. Na koncu samo še uredimo, da se v sporočilnem oknu izpiše spremenljivka `izpis`.

```
private void gPrikazi_Click(object sender, EventArgs e)
    {
        string izpis = "";
        foreach (var oznacenaDrzava in isDrzave.SelectedItems)
        {
            izpis = izpis + oznacenaDrzava.ToString() + ", ";
        }
        MessageBox.Show("Iz seznama ste izbrali: " + izpis.Substring(0, izpis.Length - 2));
    }
```

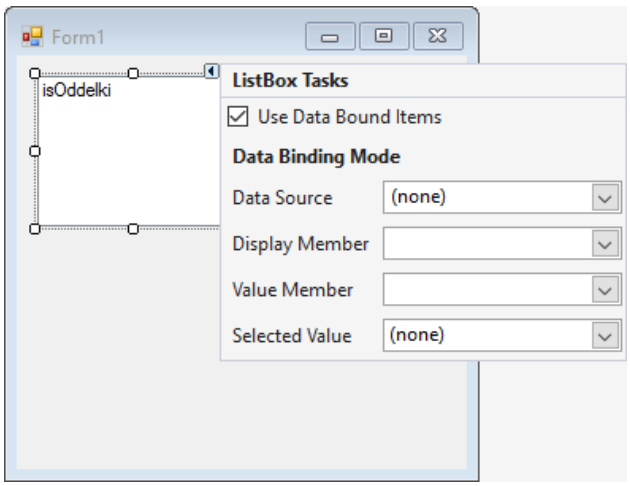
Podatke v gradniku `ListBox` lahko pridobimo tudi iz podatkovne baze. Ko odložimo gradnik `ListBox`, v desnem zgornjem kotu poiščemo trikotnik, na katerega kliknemo. Ob kliku nanj se nam pokaže okno, kot je

prikazano na sliki 84. Na njej imamo potrditveno polje, kjer izberemo, da so podatki, ki naj jih vsebuje gradnik, shranjeni drugje (v podatkovni bazi, seznamu …).



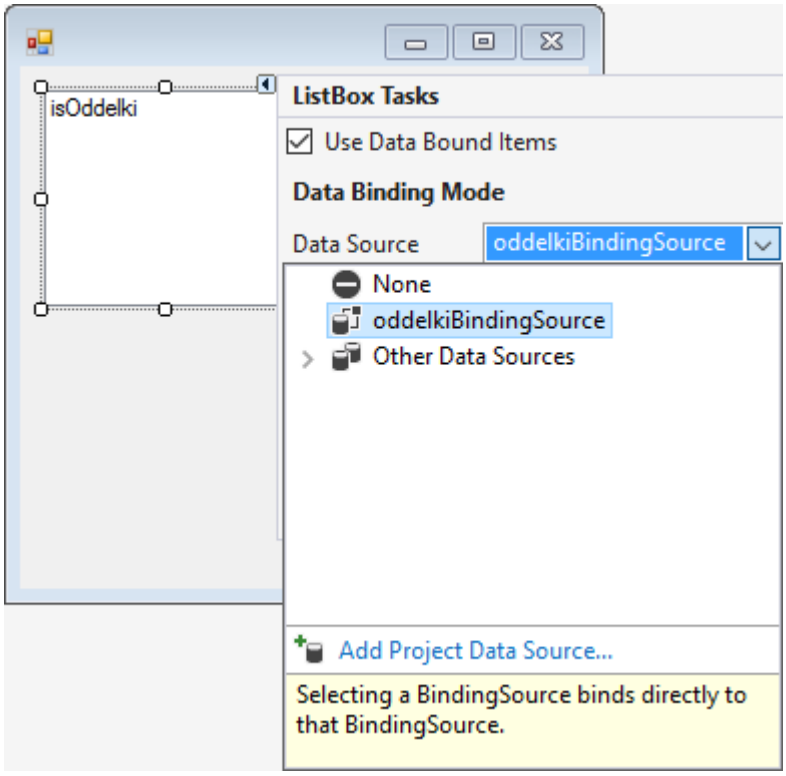
Slika 84: Navezava na vir podatkov

Poglejmo najprej, kako se navežemo na bazo.



Slika 85: Dodajanje gradnikov preko povezave

Na sliki 85 vidimo, da moramo najprej določiti, kje je vir podatkov, od koder bo gradnik črpal podatke. V spustnem polju `Data Source` kliknemo na povezavo `Add Project Data Source` … (slika 86), nastavimo povezavo do podatkovne baze (`Data Source`) in določimo, kateri stolpec tabele (`Display Member`) bomo prikazali v gradniku.

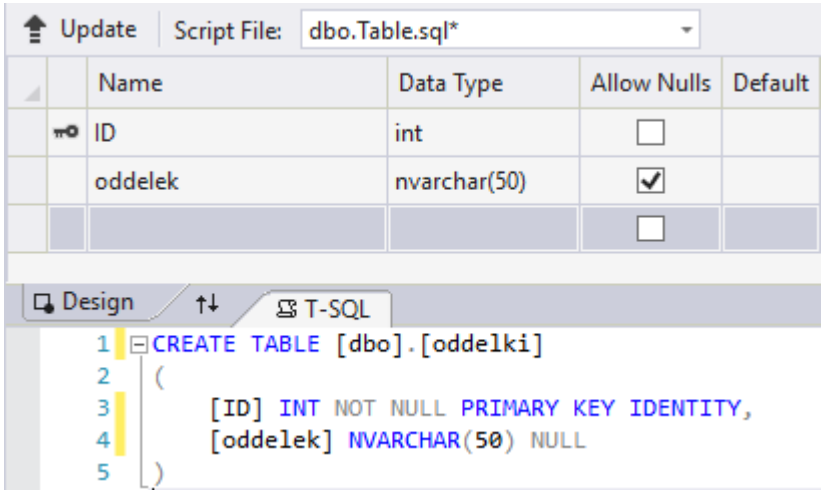


Slika 86: Vzpostavitev povezave, npr. do podatkovne baze

Na ta način napolnjen gradnik vedno kaže tak seznam, kot je na voljo v podatkovnem viru, s katerim je povezan.

Da smo to lahko storili, smo morali v projekt dodati ustrezno bazo.

Ta je lahko obstoječa, lahko pa ustvarimo tudi novo. V projekt dodamo novo bazo tako, da v Solution Explorerju kliknemo na ime projekta z desnim gumbom. Pri tem se nam pokažejo različne možnosti. Med njimi izberemo `Add` in nato `New Item` … Nato v oknu, ki se nam odpre, poiščemo tip datoteke `Service-based Database`, ga izberemo in poljubno poimenujemo ter kliknemo na gumb `Add`. V raziskovalcu rešitve poiščemo novo ustvarjeno bazo (npr. `NasaBaza.mdf`). Z dvojnim klikom nanjo se nam odpre zavihek `Server Explorer`, v katerem bazi dodamo tabele. To storimo tako, da s klikom na trikotnik poleg `NasaBaza.mdf`, razširimo izbor. Z desnim klikom na `Tables`, se nam odpre okno, v katerem izberemo `Add New Table`. Odpre se nam novo okno, v katerem določimo zgradbo tabele. V podrobnosti se ne bomo spuščali. Medtem ko te stvari nastavljamo, se nam samodejno ustvari SQL stavek, ki ustvari tako tabelo (glej sliko 87).



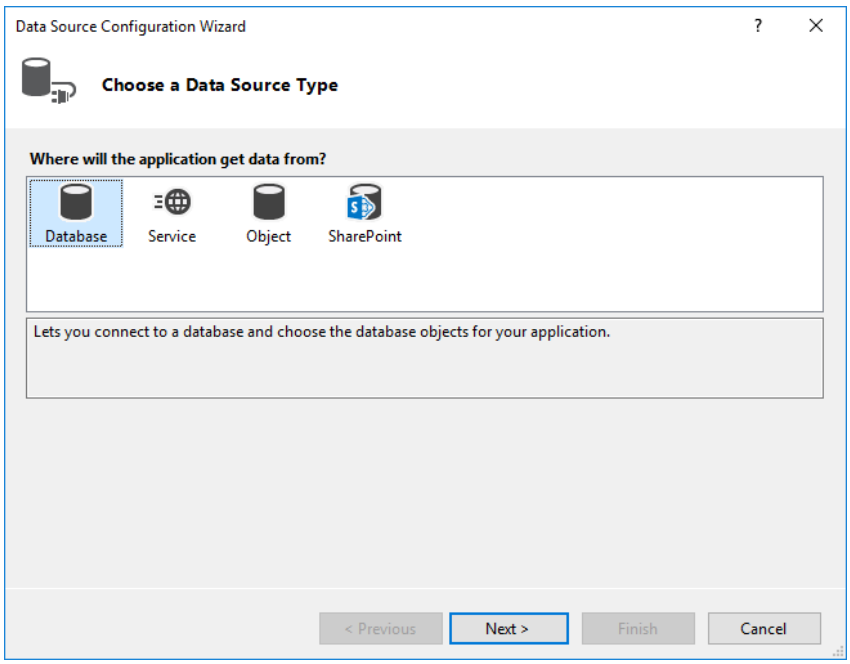
Slika 87: Ustvarjanje tabele v podatkovni bazi v okolju VS

Ko končamo z vsemi nastavitvami, kliknemo na gumb `Update`, ki nam vse novo ustvarjene nastavitve shrani in posodobi zgradbo nove tabele. V tabelo dodamo podatke tako, da odpremo zavihek `Server Explorer`, kjer se z desnim klikom na novo tabelo (`oddelki`) odpre okno, v katerem izberemo `Show Table Data`. V oknu, ki se nam odpre, nato dodamo nekaj podatkov. V našem primeru smo dodali nekaj oddelkov podjetja (slika 88).

Id	oddelek
1	uprava
2	tehnika
3	prevoz
4	komerciala
NULL	NULL

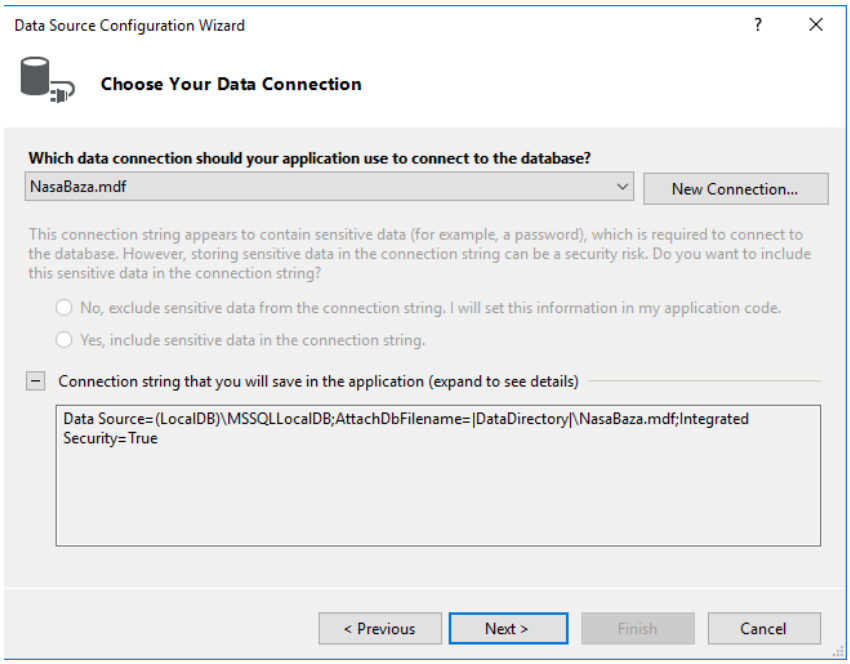
Slika 88: Dodajanje podatkov v tabelo

Sedaj lahko nastavimo gradniku `ListBox` vir, od koder bo črpal podatke, kot je prikazano na sliki 86. To naredimo tako, da kliknemo na trikotnik v zgornjem desnem kotu gradnika `ListBox`, kjer se nam odpre okno. Ko kliknemo na izbirno polje `Use Data Bound Items`, se nam to okno razširi. V spustnem polju s klikom na `Add Project Data Source` odpremo novo okno, v katerem izberemo, od kod bo gradnik dobil podatke (glej sliko 89).

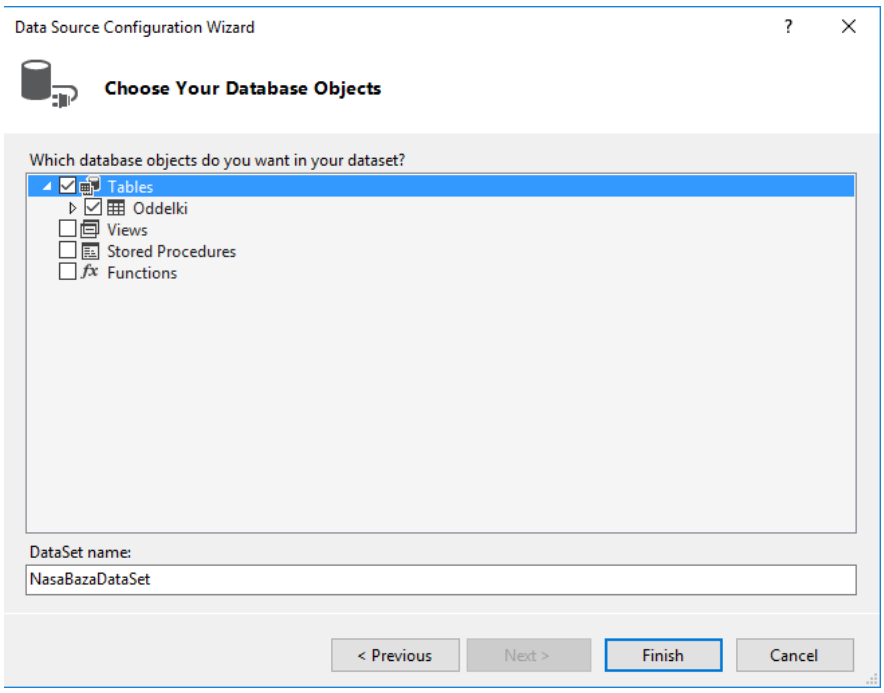


Slika 89: Izbira vira podatkov

Ker smo prej ustvarili podatkovno bazo, izberemo v tem oknu `Database` in kliknemo na gumb `Next`. Ko se nam pojavi novo okno, zopet kliknemo na gumb Next. Tu potrdimo, da se ustvari prazen objekt `Dataset`, v katerega nato napolnimo podatke iz baze. Nato se pokaže novo okno (slika 90). V njem določimo, iz katere podatkovne baze bo gradnik črpal podatke in kliknemo na gumb `Next`. V našem primeru smo izbrali bazo `NasaBaza.mdf`. Nato se nam pokaže okno, v katerem shranimo informacijo o povezavi v ustrezno konfiguracijsko datoteko. Na koncu samo še izberemo, katere objekte podatkovne baze bomo dodali v objekt `Dataset`, ki smo ga ustvarili in kliknemo na gumb `Finish` (slika 91).

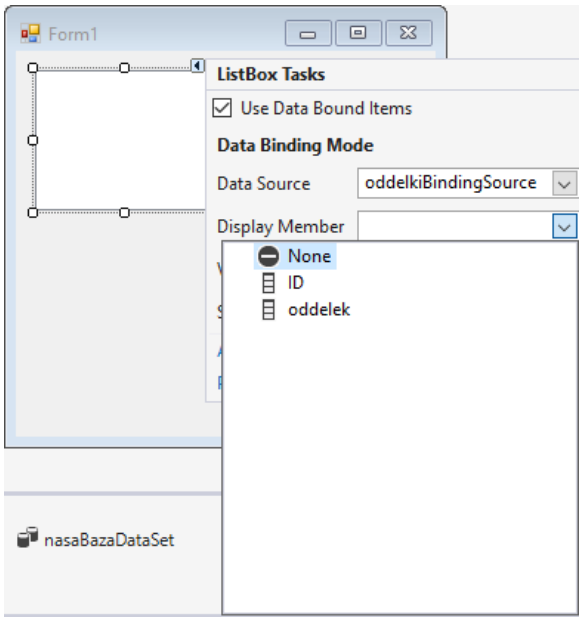


Slika 90: Ustvarjanje povezave do baze



Slika 91: Izbira objektov, ki jih želimo v našem objektu NasaBazaDataSet

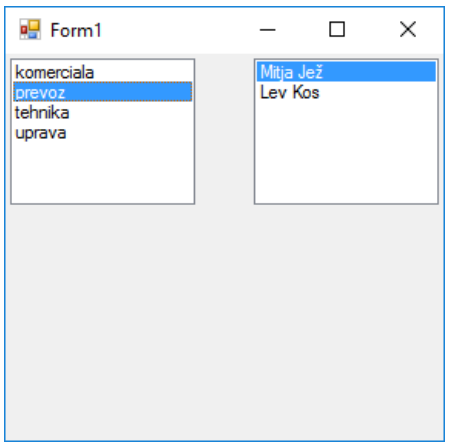
Ko uredimo, od kod gradnik dobi podatke, določimo še, katere podatke naj nato prikazuje. To naredimo tako, da v spustnem oknu na sliki 92 izberemo enega izmed stolpcev baze (npr. `oddelek`). Ko zaženemo aplikacijo, se gradnik napolni s podatki iz baze.



Slika 92: Izbira podatkov iz stolpcev tabele, ki so vidni na gradniku

Najpogosteje uporabljen dogodek gradnika `ListBox` je `SelectedIndexChanged`. Ta dogodek se zgodi, ko uporabnik označi podatek v seznamu.

Oglejmo si sedaj še primer, ko kot vir podatkov izberemo tako podatke iz baze kot tudi iz seznama. Na sliki 93 imamo v prvem izbirnem seznamu oddelke v podjetju, ki jih dobimo iz baze, ki smo jo ustvarili (`NasaBaza.mdf`). Ko uporabnik označi oddelek iz seznama, se nam v drugem pojavijo zaposleni (ime in priimek), ki delajo na teh oddelkih. Te zaposlene pa dodamo v seznam v kodi. Za to, da se ob spremembi označenega oddelka spremenijo v izbirnem seznamu zaposleni, si pomagamo z dogodkom `SelectedIndexChanged`.



Slika 93: Ob izbiri podatka iz prvega ListBoxa se nam zapišejo ustrezni podatki v drugi ListBox

Poglejmo si ustrezno kodo.

```
List<string> seznamKom = new List<string>();
List<string> seznamUpr = new List<string>();
List<string> seznamPre = new List<string>();
List<string> seznamTeh = new List<string>();

public Form1()
{
    InitializeComponent();


    seznamKom.Add("Jože Novak");
    seznamKom.Add("Stane Kralj");
    seznamKom.Add("Monika Žaba");

    seznamPre.Add("Mitja Jež");
    seznamPre.Add("Lev Kos");

    seznamTeh.Add("Miha Grič");
    seznamTeh.Add("John Frusciante");
    seznamTeh.Add("Richard Wright");

    seznamUpr.Add("Klemen Sova");
}

private void Form1_Load(object sender, EventArgs e)
{
    this.oddelkiTableAdapter.Fill(this.nasaBazaDataSet.Oddelki);

}

private void isOddelki_SelectedIndexChanged(object sender, EventArgs e)
{
    try
    {
        isZaposleni.Sorted = true;

        if ((isOddelki.SelectedItem as DataRowView)["oddelek"].ToString() == "komerciala")
        {
            isZaposleni.DataSource = seznamKom;
        }
        else if ((isOddelki.SelectedItem as DataRowView)["oddelek"].ToString() == "prevoz")
        {
            isZaposleni.DataSource = seznamPre;
        }
        else if ((isOddelki.SelectedItem as DataRowView)["oddelek"].ToString() == "tehnika")
        {
```

```
            isZaposleni.DataSource = seznamTeh;
        }
        else if ((isOddelki.SelectedItem as DataRowView)["oddelek"].ToString() == "uprava")
        {
            isZaposleni.DataSource = seznamUpr;
        }
        else
        {
            MessageBox.Show("Dodan je bil nov oddelek, ki še ni predviden v aplikaciji.");
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

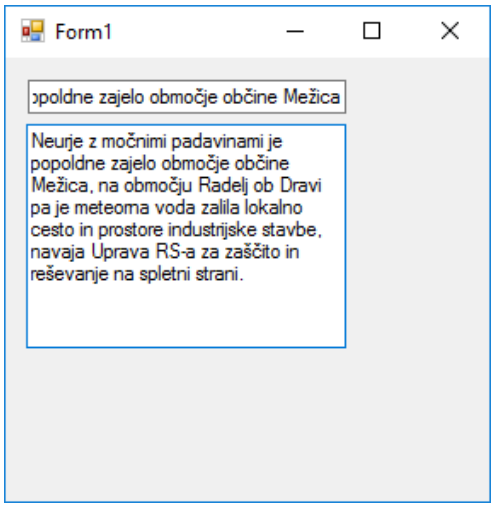
Vidimo, da smo v izbirni seznam `isOddelki` vstavili oddelke, ki smo jih dodali v bazi. Ko določimo, iz katerega vira nam gradnik prikazuje podatke, se nam samodejno ustvari koda, zapisana v metodi `Form1_Load`. V vsakega izmed oddelkov, ki smo jih ustvarili na začetku (`seznamKom`, `seznamTeh`, `seznamUpr`, `seznamPre`), smo nato dodali zaposlene, ki delajo na tem oddelku. Kadar uporabnik iz levega seznamskega polja izbere npr. oddelek uprava, se v desnem seznamskem polju pojavijo imena in priimki vseh oseb, ki so zaposlene na tem oddelku. Ob tem tudi osebe znotraj oddelka razporedimo po abecednem redu imen.

Da smo izbirni seznam `isZaposleni` napolnili z ustreznimi podatki, smo poskrbeli tako, da smo lastnost `DataSource` nastavili na ustrezni seznam.

Za več informacij o gradniku si poglejte [27].

TextBox /Vnosno polje

Gradnik TextBox uporabimo v aplikaciji, ko želimo, da uporabnik vpiše določene podatke. V osnovi je gradnik uporabljen za pregled ali vpis vhodnih podatkov v eni vrstici. Ustrezno besedilo je v lastnosti `Text`. Če želimo, da je vnosno polje večvrstično, to uredimo preko lastnosti `Multiline`, ki jo nastavimo na `True` (slika 88). Za več informacij o gradniku poglejte [28].

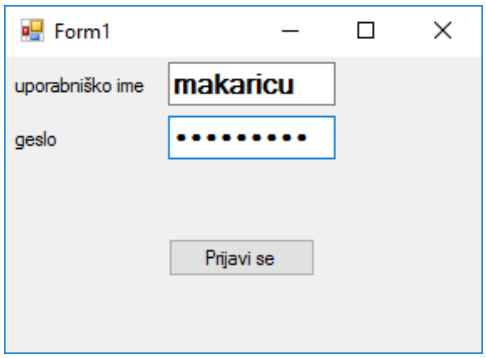


Slika 94: Zapis znotraj enovrstičnega in večvrstičnega vnosnega polja

Kot je razvidno s slike 94 v prvem vnosnem polju, ne vidimo celotnega vnesenega besedila. To lahko uredimo tako, da znotraj gradnika omejimo število vnesenih znakov. To nastavimo s pomočjo lastnosti

`MaxLength`, kjer določimo največje možno število znakov. Privzeto je nastavljeno, da lahko uporabnik vpiše 32.767 znakov.

Vnosno polje lahko nastavimo tudi tako, da je zapis znotraj njega skrit. To urejamo tako, da v lastnosti `PasswordChar` uporabimo poljuben znak. Ta je potem viden namesto besedila. To lastnost uporabimo, ko delamo npr. prijavni obrazec, v katerega se mora uporabnik vpisati s svojim uporabniškim imenom in geslom.



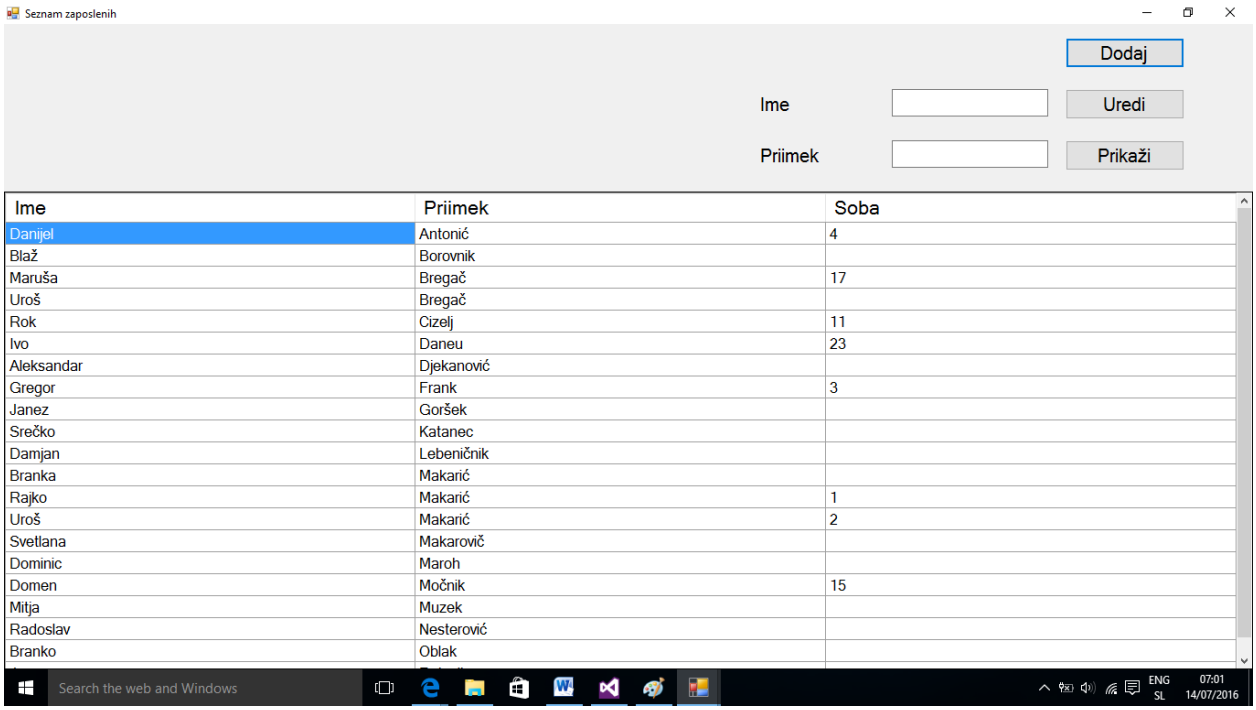
Slika 95: Primer prijavnega obrazca s skritim besedilom v besedilnem polju gesla

Pri tem smo uporabili naslednjo kodo, ki ne potrebuje dodatne razlage

```
public Form1()
{
    InitializeComponent();
    FontFamily imePisave = new FontFamily("Microsoft Sans Serif");
    Font stil = new Font(imePisave, 12, FontStyle.Bold);
    bpUporabnik.Font = stil;
    bpGeslo.Font = stil;
    bpGeslo.PasswordChar = '•';
    gPotrdi.Text = "Prijavi se";
}
```

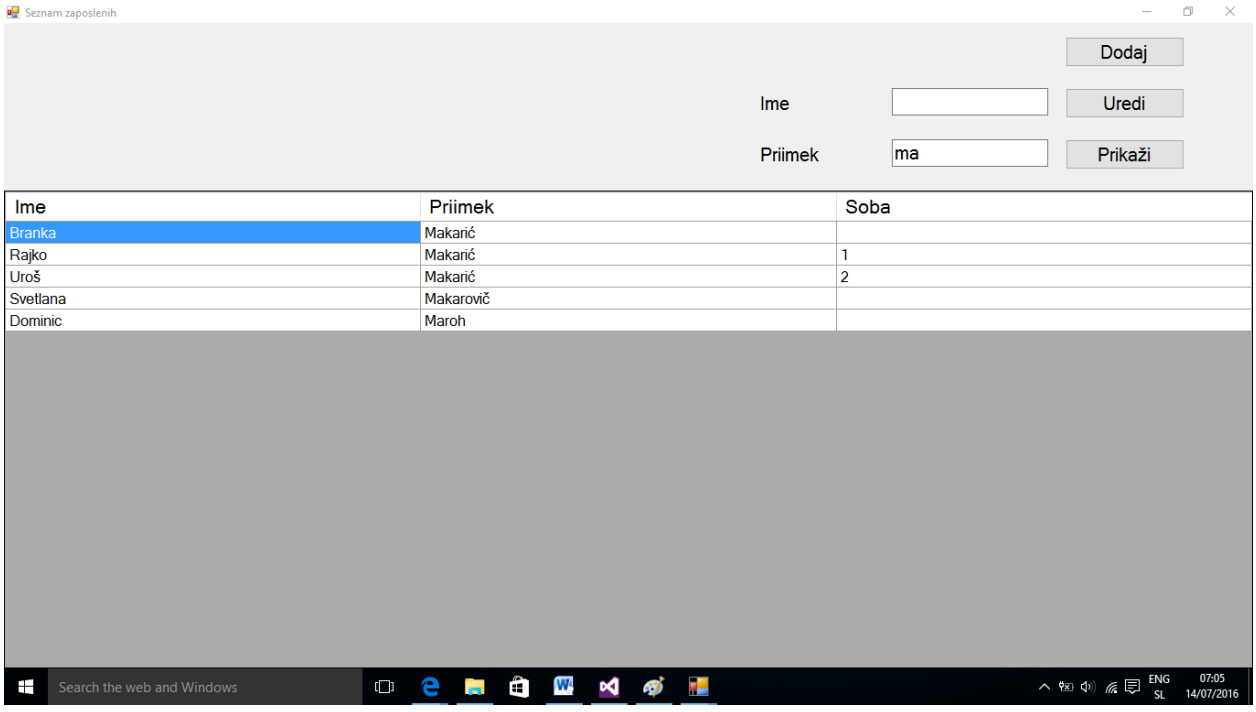
Pri gradniku `TextBox` najpogosteje uporabimo dogodek `TextChanged`, ki se sproži vsakič, ko se besedilo znotraj polja spremeni. Oglejmo si uporabo, ko preko vnosa v tekstno polje filtriramo izpis seznama, ki ga dobimo iz podatkovne baze. Pri tem primeru se bomo omejili na stvari v povezavi z gradnikom `TextBox` in bomo predpostavili, da ima bralec že izkušnje z bazami v VS in gradnikom `DataSet`.

Na sliki 96 imamo prikazano aplikacijo, ki nam na začetku prikaže vse podatke iz podatkovne baze. Ko pa uporabnik vpiše v eno izmed vnosnih polj nekaj znakov, se začne ta seznam manjšati. Tako lahko uporabnik hitro najde podatke, ki jih potrebuje.



Slika 96: Aplikacija, ki nam na začetku prikazuje vse podatke iz baze

Na sliki 97 vidimo, da se, ko uporabnik v vnosno polje priimka vpiše črko "ma", seznam zmanjša in prikaže samo tiste, kjer se priimek v seznamu začne s tema črkama. Če bi uporabnik zopet spremenil in iz vnosnega polja pobrisal ti dve črki, bi se seznam povečal na prvotno obliko.



Slika 97: Z zapisom v besedilnem polju se spremeni pregled podatkov gradnika DataGridView

Spodaj si oglejmo kodo, ki nam prikaže začetni seznam in njegovo spremembo v primeru vpisa besedila v vnosno polje.

```
DataSet ds = new DataSet(); // ustvarimo nov objekt tipa DataSet za shranjevanje podatkov v
pomnilniku
SqlConnection povezava = new SqlConnection(@"Data Source=|DataDirectory|\Baza.sdf"); // kre-
iramo povezavo do baze na katero se povežemo
```

```
SqlDataAdapter da = new SqlDataAdapter(); // ustvarimo objekt tipa SqlDataAdapter za komuni-
kacijo s tabelo podatkov iz baze

private void Form1_Load(object sender, EventArgs e)
{
    da.SelectCommand = new SqlCommand("SELECT ime as Ime, priimek AS Priimek, st_sobe AS
Soba FROM zaposleni ORDER BY priimek, ime ASC", povezava); // ustvarimo poizvedbo iz baze
    ds.Clear(); // pobrišemo vse iz objekta ds če so slučajno že kakšni podatki
    da.Fill(ds); // dodamo oz. osvežimo vrstice ds, s podatki iz poizvedbe
    ppZaposleni.DataSource = ds.Tables[0]; //
}

private void vpisPriimka_TextChanged(object sender, EventArgs e)
{
    DataRow[] vrsticeIme = ds.Tables[0].Select(string.Format("ime LIKE '{0}*'", vpisIme-
na.Text)); // vse vrstice iz objekta ds, kjer je ime podobno kot v vnosnem polju
    DataRow[] vrsticePriimek = ds.Tables[0].Select(string.Format("priimek LIKE '{0}*'", vpi-
sPriimka.Text)); // vse vrstice iz objekta ds, kjer je priimek podoben kot v vnosnem polju

    DataTable novaTabela = new DataTable(); // ustvarimo novo tabelo
    novaTabela = ds.Tables[0].Clone(); // prekopiramo strukturo tabele iz objekta ds

    foreach (DataRow vrsticaPriimka in vrsticePriimek) // preverimo vsak priimek iz vrstice
vrsticePriimek
    {
        foreach (DataRow vrsticaImena in vrsticeIme) // preverimo vsako ime iz vrstice vrs-
ticeIme
        {
            if (vrsticaPriimka[0].Equals(vrsticaImena[0]) && vrsticaPriim-
ka[1].Equals(vrsticaImena[1]) && vrsticaPriimka[2].Equals(vrsticaImena[2])) // preverimo ali
so stolpci vrstic vrsticeImena in vrsticePriimka enake
            {
                DataRow novaVrstica = NewTable.NewRow(); // če so vrstice enake v tabelo
dodamo to vrstico
                novaVrstica[0] = vrsticaImena[0]; // kjer je prvi stolpec vrsticeImena tudi
prvi stolpec vrstice r
                novaVrstica[1] = vrsticaImena[1]; // drugi stolpec vrsticeImena drugi vrsti-
ce r
                novaVrstica[2] = vrsticaImena[2]; // in podobno tretji stolpec
                novaTabela.Rows.Add(novaVrstica); // ter dodamo vrstico v tabelo
            }
        }
    }
    ppZaposleni.DataSource = novaTabela; // na koncu na gradniku ppZaposleni prikažemo poda-
tke iz tabele novaTabela
}
```

Najprej ustvarimo objekte `DataSet`, `SqlDataAdapter` in `SqlConnection`. V objektu `SqlConnection` zapišemo, s katero podatkovno bazo se povežemo in pokažemo pot do izvora podatkov, s katerimi bomo v nadaljevanju delali. `SqlDataAdapter` skrbi za komunikacijo med aplikacijo in podatkovno bazo. Z njim poleg tega, da se povežemo z bazo, tudi izvedemo SQL poizvedbe, shranjujemo podatke v `DataSet` in jih zapišemo v podatkovno bazo. Objekt `DataSet` pa uporabljamo za shranjevanje podatkov, ki jih dobimo s pomočjo `SqlDataAdapter`, v pomnilnik. Ko imamo vse te objekte pripravljene, z njimi na gradniku `DataGridView` prikažemo tabelo iz podatkovne baze. Ob zagonu aplikacije, ko sta vnosni polji prazni, v gradniku `DataGridView` prikažemo vse podatke iz tabele `zaposleni`. Ko začnemo v vnosno polje priimka pisati, se sproži metoda `vpisPriimka_TextChanged`. V njej imamo zapisano, da se ob vsaki spremembi v vnosnem polju v gradnik `DataGridView` izpišejo tisti rezultati, ki v vnosnem polju ustrezajo imenom in priimkom v tabeli baze. Tako se ob vnosu znakov v vnosno polje v `vrsticeIme` shranijo samo določene vrstice iz objekta `DataSet`, kjer je shranjena tabela zaposleni iz baze. Podobno naredimo tudi s priimkom. Nato ustvarimo novo tabelo, v katero prekopiramo strukturo iz

tabele `zaposleni`. Nato pogledamo vse vrstice `vrsticePriimek` in `vrsticeIme` in preverimo, ali se ujemajo stolpci posamezne vrstice `vrsticePriimek` in `vrsticeIme`. Tiste vrstice, ki so enake, nato shranimo v `novaVrstica`, ki jo shranimo v `novaTabela`. Na koncu prikažemo na gradniku `DataGridView` podatke iz `novaTabela`, ki jo predstavimo kot podatkovni vir, ki prikazuje zadetke.

V podrazdelku `Data` si bomo podrobneje ogledali lastnosti gradnika `DataGridView`, v katerem prikazujemo tabelo `zaposleni` iz podatkovne baze.

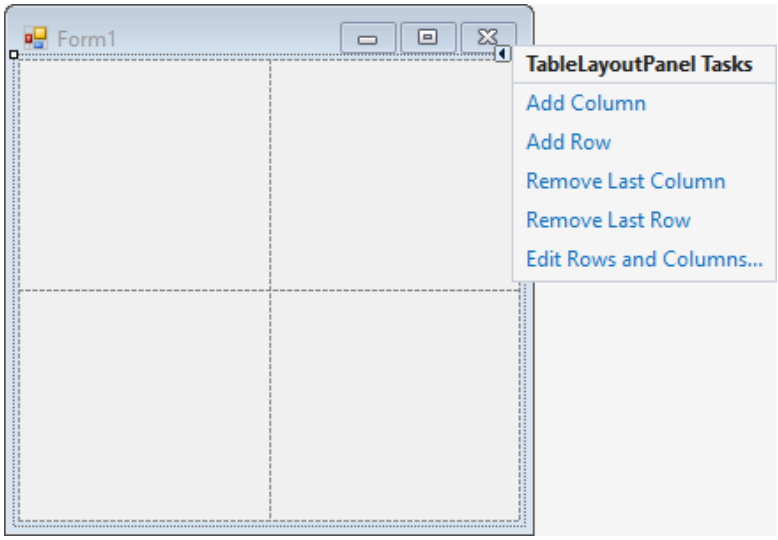
Containers – Vsebniki

To so gradniki, ki jih uporabimo za to, da vanje dodamo druge gradnike. Znotraj vsebnikov je običajno vsebovanih več različnih gradnikov, ki jih lahko urejamo. Običajno uporabljamo vsebnike zato, da vanje dodamo skupino gradnikov, ki so na nekakšen način med seboj povezani. Ta povezava je lahko odvisna od položaja na obrazcu ali kot neka skupina izbir (npr. primer gradnika `RadioButton` na strani 57) ali kako drugače.

Zelo uporabno pri vsebnikih je to, da lahko položaj in velikost gradnikov določimo glede na vsebnik in tako lažje sočasno prestavljamo vse gradnike v skupini. To naredimo tako, da skupino gradnikov označimo in jo z miško prenesemo na drugo mesto na obrazcu.

TableLayoutPanel/Razpored gradnikov v tabelarični obliki

Predstavlja dinamično pregledno ploščo, ki prikaže vsebino okna v mrežasti obliki vrstic in stolpcev.



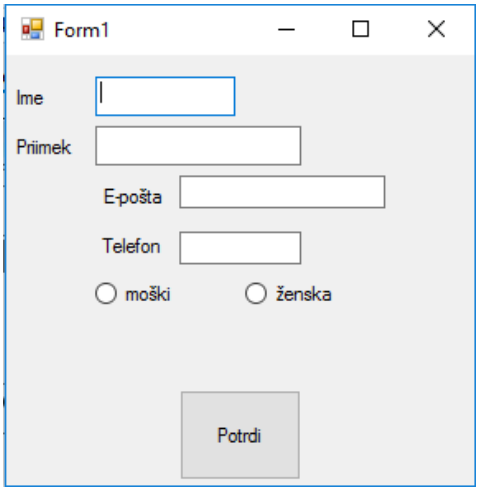
Slika 98: Gradnik TableLayoutPanel

Gradnik `TableLayoutPanel` razdeli površino, ki jo zavzema, na določeno število stolpcev in vrstic. Privzeto imamo 4 celice (dve vrstici in dva stolpca). V te celice bomo potem zlagali druge gradnike. Če želimo dodati ali izbrisati kakšen stolpec ali vrstico, to naredimo tako, da kliknemo na trikotnik v desnem zgornjem kotu gradnika in nato še `Edit Rows and Columns ...` (slika 98). Po kliku se nam odpre okno, v katerem lahko nato urejamo število vrstic, stolpcev, njihovo višino in širino.

V celice gradnike dodamo z metodo `Controls.Add`., kjer zapišemo, kateri gradnik dodajamo in v katero celico. Običajno lastnost `Dock` gradnika, ki ga dodamo v `TableLayoutPanel`, nastavimo na `DockStyle.Fill`, kar pomeni, da dodani gradnik zavzame celotno vsebino celice. Če pa želimo določeni gradnik razporediti preko več stolpcev in/ali vrstic, pa to naredimo z lastnostma `SetColumnSpan` ter `SetRowSpan`.

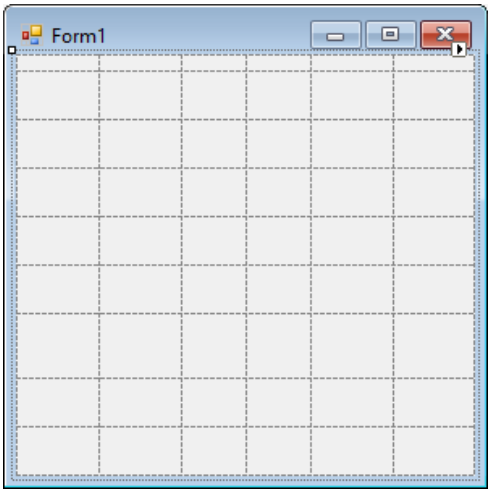
Več informacij o gradniku najdete [29].

Poglejmo si zgled. Denimo da želimo ustvariti tak videz uporabniškega vmesnika, kot je prikazan na sliki 99.



Slika 99: Obrazec, narejen s pomočjo gradnika TableLayoutPanel

Najlažje bomo to storili, če si predstavljamo, da imamo površino okna, razdeljeno z mrežo 6 x 9, kar smo označili na sliki 100.



Slika 100: Razporeditev vrstic in stolpcev gradnika

Če primerjamo sliki 100 in 99, vidimo, da smo določene gradnike razpotegnili preko dveh ali več stolpcev in/ali vrstic. Tako smo gradnik `TextBox`, kamor uporabnik vpiše ime, razširili na 2 celici in gradnik, kamor vpiše priimek, na 3 celice. Podobno kot smo naredili za ime in priimek, naredimo tudi za vnosni polji, kamor uporabnik vpiše elektronsko pošto in telefon. Radijska gumba pa tudi razpotegnemo preko dveh stolpcev v 6. vrstici. Poglejmo še gumb na sliki 99, kjer vidimo, da smo gradnik razpotegnili na dve vrstici in dva stolpca. Samo oznakam smo pustili, da zasedejo le celico, v katero smo jih odložili.

Iz orodjarne torej potegnemo gradnik `TableLayoutPanel`, ga preimenujemo v `gtoVpis` in mu dodamo sedem vrstic ter štiri stolpce. Nato na gradnik `gtoVpis` razporedimo gradnike `oIme`, `oPriimek`, `oMail`, `oTel`, `vpIme`, `vpPriimek`, `vpMail`, `vpTel`, `rgMoski`, `rgZenska` in `gPotrdi`. Ostale spremembe, ki smo jih prej opisali, opravimo s kodo. Spodaj si oglejmo ustrezno kodo.

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

            //gradniki v tabelarični obliki
```

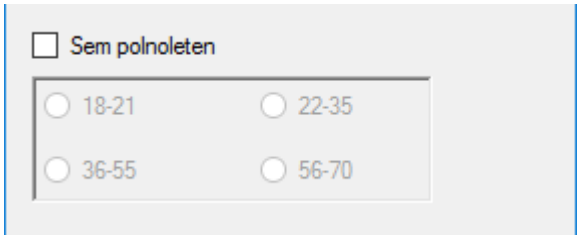
```
        gtoVpis.Controls.Add(oIme, 0, 1);
        gtoVpis.Controls.Add(oPriimek, 0, 2);
        gtoVpis.SetColumnSpan(gPotrdi, 2);
        gtoVpis.Controls.Add(gPotrdi, 2, 7);
        gtoVpis.SetColumnSpan(vpIme, 2);
        gtoVpis.Controls.Add(vpIme, 1, 1);
        gtoVpis.SetColumnSpan(vpPriimek, 3);
        gtoVpis.Controls.Add(vpPriimek, 1, 2);
        gtoVpis.SetColumnSpan(vpMail, 3);
        gtoVpis.SetColumnSpan(vpTel, 2);
        gtoVpis.SetColumnSpan(rgMoski, 2);
        gtoVpis.SetColumnSpan(rgZenska, 2);

        //oznaka ime
        oIme.Dock = DockStyle.Fill;
        oIme.TextAlign = ContentAlignment.MiddleLeft;
        oIme.Text = "Ime";

        //oznaka priimek
        oPriimek.Dock = DockStyle.Fill;
        oPriimek.TextAlign = ContentAlignment.MiddleLeft;
        oPriimek.Text = "Priimek";

        //vnosno polje ime
        vpIme.Dock = DockStyle.Fill;
        vpIme.Multiline = true;

        //vnosno polje priimek
        vpPriimek.Dock = DockStyle.Fill;
        vpPriimek.Multiline = true;

        //oznaka e-mail
        oMail.Dock = DockStyle.Right;
        oMail.TextAlign = ContentAlignment.MiddleCenter;
        oMail.Text = "E-pošta";

        //vnosno polje e-mail
        vpMail.Dock = DockStyle.Fill;
        vpMail.Multiline = false;

        //oznaka telefon
        oTel.Dock = DockStyle.Fill;
        oTel.TextAlign = ContentAlignment.MiddleCenter;
        oTel.Text = "Telefon";

        //vnosno polje telefon
        vpTel.Dock = DockStyle.Bottom;
        vpTel.Multiline = false;

        //gumb
        gPotrdi.Dock = DockStyle.Fill;
        gPotrdi.Text = "Potrdi";

        //radio gumb moški
        rgMoski.Text = "moški";
        rgMoski.Dock = DockStyle.Fill;

        //radio gumb ženska
        rgZenska.Text = "ženska";
        rgZenska.Dock = DockStyle.Fill;
    }
```

Panel/Plošča

To je gradnik, na katerega odlagamo druge gradnike. Po navadi uporabimo ta gradnik, da ustvarimo skupino gradnikov, kot je npr. `RadioButton`, kar smo spoznali že v razdelku o radijskem gumbu. Gradnik `Panel` je privzeto nastavljen brez obrobe, vendar lahko to spremenimo z lastnostjo `BorderStyle`. Tako lahko grafično ločimo ta gradnik od preostalih delov obrazca. Pri tem imamo 3 možnosti. Prva je privzeta, kjer je vrednost lastnosti `None`. Če želimo, da imamo ploščo od ostalega obrazca ločeno, spremenimo vrednost lastnosti `BorderStyle` na `FixedSingle` ali `Fixed3D`. Kot pri `TableLayoutPanel`, imamo tudi tu lastnost `Controls`, s katero lahko dodajamo, ali odstranimo gradnike na plošči.

Uporabna lastnost tega gradnika je `Enabled`, s katero določamo, ali je gradnike, ki so na plošči, možno uporabljati. V primeru, da ima ta lastnost vrednost `False`, uporabnik gradnike znotraj vsebnika vidi, vendar so vsi zaklenjeni in jih ne more uporabljati. Več informacij o gradniku dobite v [30].

Poglejmo si primer, kjer nastavimo, da se plošča in gradniki, vsebovani v njej, odklenejo, ko uporabnik izbere določeno vrednost.



Slika 101: Gradnik Panel zaklenjen

Ko uporabnik označi potrditveno polje, se odklene vsebnik in šele nato lahko označi tudi enega izmed radijskih gumbov.

```
public Form1()
{
    InitializeComponent();
    pZaklenjena.Enabled = false;
    pZaklenjena.BorderStyle = BorderStyle.Fixed3D;

    ppPrikazi.Text = "Sem polnoleten";

    rgStar21.Text = "18-21";
    rgStar35.Text = "22-35";
    rgStar55.Text = "36-55";
    rgStar70.Text = "56-70";
}

private void ppPrikazi_CheckedChanged(object sender, EventArgs e)
{
    if (ppPrikazi.Checked == true)
    {
        pZaklenjena.Enabled = true;
    }
    else
    {
        pZaklenjena.Enabled = false;
    }
}
```

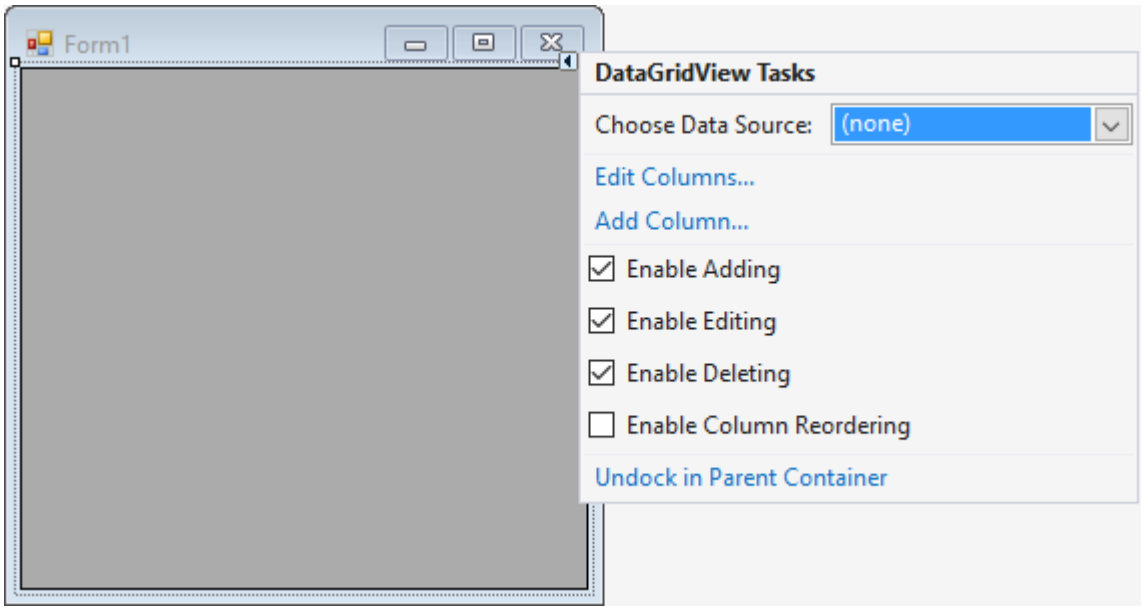
V kodi vidimo, da smo na začetku nastavili, da je gradnik `Panel` ob zagonu aplikacije zaklenjen. Poleg tega smo nastavili tudi obrobo gradnika. Gradniku `CheckBox` smo nastavili dogodek, ki preveri, ali je potrditveno polje označeno. V primeru, da je, lahko uporabnik iz plošče izbere, v kateri starostni skupini je.

Data – Podatki

V tej skupini imamo gradnike, ki jih uporabimo za delo s povezavami do določenih podatkov (podatkovne baze, seznami …). Ogledali si bomo le gradnik `DataGridView`.

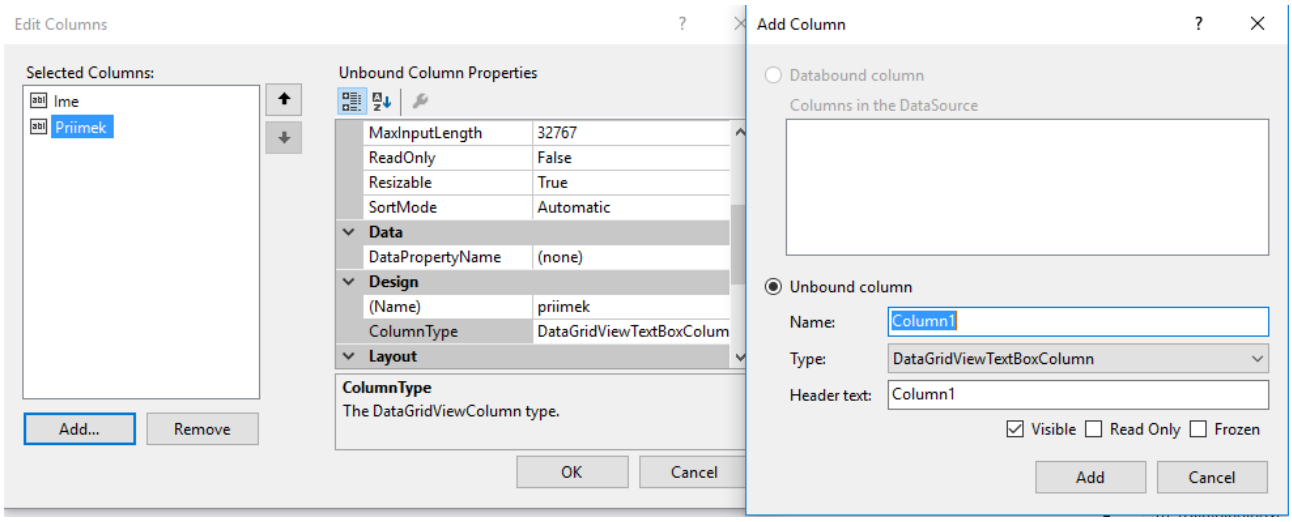
DataGridView/Pregled podatkov v obliki tabele

`DataGridView` nam omogoča pregled podatkov v obliki tabele. Gradnik omogoča prilagodljivost celic, vrstic, stolpcev in obrob s pomočjo različnih lastnosti, kot so `DefaultCellStyle`, `ColumnHeadersDefaultCellStyle`, `CellBorderStyle` in `GridColor`.



Slika 102: Gradnik DataGridView na obrazcu z oknom DataGridView Tasks

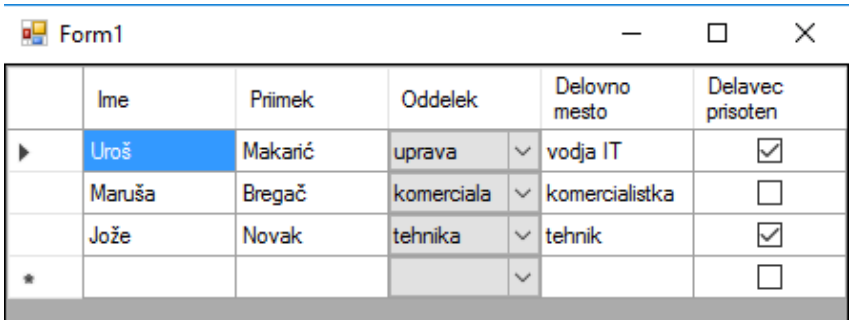
Ko odložimo gradnik na obrazec, se nam ob njem pojavi okno, v katerem nastavimo določene lastnosti (slika 102). Tu nastavimo, ali uporabniku omogočimo dodajanje, urejanje in brisanje podatkov iz gradnika. Poleg tega tu lahko določimo, da so podatki iz gradnika prikazani iz nekega zunanjega vira, kot je npr. podatkovna baza (spomnimo se na gradnik `ListBox` s strani 61). Ko izberemo v oknu `DataGridView Tasks` možnost `Edit Columns` …, se nam odpre okno, kot na sliki 103.



Slika 103: Ustvarjanje stolpcev gradnika

Tu ustvarimo stolpce in določimo nekaj osnovnih lastnosti gradnika, kot so `DefaultCellStyle`, `Width`, `AutoSizeMode` … V lastnosti `DefaultCellStyle` nastavimo stil celic gradnika. To pomeni, da uredimo slog pisave, velikost besedila, barvo ozadja celice, postavitev elementov znotraj celice … V lastnosti `Width` nastavimo širino stolpca, z lastnostjo `AutoSizeMode` pa uredimo, kako se širina stolpca prilagodi znotraj gradnika. Ko vse lastnosti nastavimo, okno, v katerem urejamo stolpce, zapremo in odkljukamo možnosti, ki jih uporabnik potrebuje pri delu. Če predvidimo, da mora uporabnik v gradnik dodajati nove vrstice, označimo potrditveno polje `Enable Adding`. Enako naredimo tudi za potrditvena polja `Enable Editing`, `Enable Deleting` in `Enable Column Reordering`, če predvidimo, da bo uporabnik moral podatke urejati, brisati, ali pa spreminjati postavitev stolpcev na gradniku.

Poglejmo si, kako dodamo podatke v novo vrstico gradnika.



Slika 104: Vpis podatkov v gradnik

Vrstica, v katero lahko dodamo nov zapis, je označena z zvezdico v naslovu vrstice. V našem primeru vrstico dodamo tako, da v stolpce `Ime`, `Priimek` in `Delovno mesto` vpišemo zahtevane podatke, označimo potrditveno okno in izberemo element iz spustnega polja (glej sliko 104). Ko izpolnimo vrstico, s klikom na tipko Enter zapis v vrstici shranimo, mi pa se prestavimo v novo prazno vrstico.

Poglejmo si še, kako bi isto vstavljanje podatkov opravili preko kode. Potrebovali bomo spustno polje. Zato moramo najprej ustvariti prazen seznam oddelkov, v katerega dodamo vse oddelke podjetja. Nato za stolpec, kjer je podatek o oddelku, kot podatkovni vir določimo seznam `seznamOddelkov`. Na koncu vpišemo zaposlene v vrstice s pomočjo metode `Rows.Add()`. Vanjo moramo vpisati toliko parametrov, kolikor imamo stolpcev, ki smo jih ustvarili (`Ime`, `Priimek`, `Oddelek`, `Delovno mesto` in `Delavec prisoten`). Pri tem kot tretji parameter izberemo eno od polj spustnega seznama (ki ga imamo v `seznamOddelkov`).

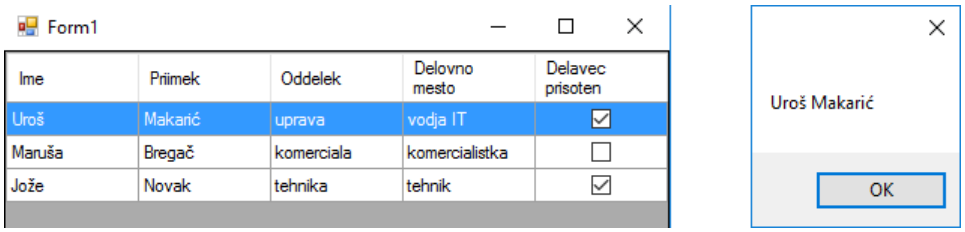
```
public Form1()
{
    InitializeComponent();
    ppZaposleni.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;

    //kreiramo seznam oddelkov
    List<string> seznamOddelkov = new List<string>();
    seznamOddelkov.Add("uprava");
    seznamOddelkov.Add("tehnika");
    seznamOddelkov.Add("prevoz");
    seznamOddelkov.Add("komerciala");

    //seznam povežemo s stolpcem oddelka
    ((DataGridViewComboBoxColumn)ppZaposleni.Columns["oddelek"]).DataSource =
         seznamOddelkov;

    //dodajanje vrstic v gradnik DatagridView
    ppZaposleni.Rows.Add("Uroš", "Makarić", seznamOddelkov[0], "vodja IT", true);
    ppZaposleni.Rows.Add("Maruša", "Bregač", seznamOddelkov[3], "komercialistka", false);
    ppZaposleni.Rows.Add("Jože", "Novak", seznamOddelkov[1], "tehnik", true);
```

}

Gradnik ima lastnost `SelectionMode`, s katero urejamo, koliko celic lahko uporabnik označi. Privzeto je nastavljena lastnost na `RowHeaderSelect`, s katero lahko uporabnik označi vsako posamezno celico, ki je na gradniku. To pa lahko spremenimo, če nastavimo lastnost `SelectionMode` na katero izmed drugih možnosti. Običajno želimo, da se nam ob izbiri določene celice, označi celotna vrstica. To naredimo tako, da lastnost `SelectionMode` nastavimo na `FullRowSelect`.

V tem primeru, ko uporabnik označi celo vrstico, je najpogosteje uporabljen dogodek `CellDoubleClick`. Ta dogodek se sproži, ko uporabnik z dvojnim klikom klikne kjerkoli v celici.

Poglejmo si primer, kjer uporabnik z dvojnim hitrim klikom odpre sporočilno okno z določenimi podatki iz vrstice gradnika `DataGridView`, v kateri smo kliknili.



Slika 105: Ob kliku na poljubno vrstico gradnika se odpre sporočilno okno s podatki iz vrstice

Ko uporabnik z miško klikne na poljubno vrstico gradnika, se zažene metoda `ppOdhod_CellClick()`, v kateri imamo zapisano navodilo, da se nam ob kliku na poljubno vrstico gradnika odpre sporočilno okno s podatki iz stolpca `Ime` in `Priimek` vrstice, v kateri uporabnik opravi klik.

Poglejmo si kodo, kjer se s klikom na poljubno celico vrstice, odpre sporočilno okno z določenimi podatki.

```
private void ppZaposleni_CellDoubleClick(object sender, DataGridViewCellEventArgs e)
{
    //poiščemo vrstico v kateri kliknemo in v spremenljivko imePriimek vstavimo besedilo iz
dveh stolpcev vrstice
    try
    {
        int vrstica = e.RowIndex;
        string imePriimek = (string)ppZaposleni[0, vrstica].Value + " "
                          + (string)ppZaposleni[1, vrstica].Value;
        MessageBox.Show(imePriimek);
    }
    catch (Exception ex)
    {
        MessageBox.Show("Prišlo je do napake!\n" + ex.Message);
    }
}
```

Z `e.RowIndex` izvemo, v kateri vrstici je uporabnik opravil klik. Nato preko lastnosti `Value` dobimo ustrezna podatka iz stolpcev `Ime` in `Priimek`. Za več informacij o gradniku si poglejte [31].

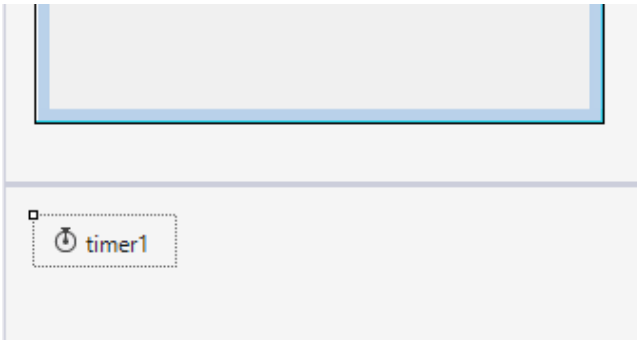
Components – Sestavni deli

V tej skupini so gradniki, s katerimi določene objekte uporabimo med aplikacijami. Ogledali si bomo le enega, in sicer `Timer`.

Timer/Časovnik

V razdelku o gradnikih smo povedali, da ločimo gradnike na tiste, ki so vidni tako v razvojnem delu kot tudi na GUV aplikacije, in na tiste, ki so vidni samo v razvojnem delu. Eden izmed teh gradnikov, ki so vidni

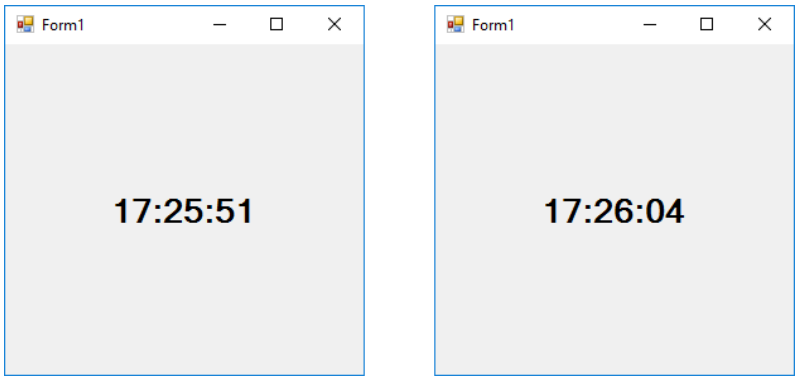
samo v razvojnem delu, je gradnik `Timer`. Gradnik uporabimo, da nam v določenih časovnih intervalih proži dogodke (dogodek `Tick`). Ta dogodek potem uporabimo, da napišemo ustrezno odzivno proceduro, ki se torej izvede vsakih določeno število milisekund. Ko gradnik odložimo na obrazcu, se pokaže v polje pod samim obrazcem (slika 106).



Slika 106: Odloženi gradnik Timer se pokaže pod obrazcem

Gradnik za razliko od preostalih gradnikov, o katerih smo govorili, nima veliko lastnosti. Dve najpomembnejši pa sta `Enabled` in `Interval`. Lastnost `Enabled` je privzeto nastavljena na `False`. To pomeni, da časovnik ne proži dogodkov. Lastnosti `Interval` določa, na koliko milisekund se sproži dogodek `Tick` (seveda takrat, ko ima lastnost `Enabled` vrednost `True`). Več informacij o gradniku si poglejte na [32].

Uporabimo ta gradnik za izdelavo enostavne ure (slika 107).



Slika 107: Enostavna ura

```
public Form1()
{
    InitializeComponent();
    uSekunda.Enabled = true;
    uSekunda.Interval = 1000;
}

private void uSekunda_Tick(object sender, EventArgs e)
{
    oUra.Text = DateTime.Now.ToLongTimeString();
}
```

V aplikaciji smo uporabili dva gradnika – oznako in časovnik. Časovnik omogočimo in nastavimo, da vsako sekundo sproži dogodek `Tick`. Kot reakcijo na ta dogodek v metodi `uSekunda_Tick()` spremenimo napis na oznaki na trenutni čas.

WPF

Sedaj si oglejmo še nekaj gradnikov, ki jih uporabljamo, ko aplikacija temelji na tehnologiji WPF.

Common WPF Controls – Pogosti WPF gradniki

Grid/Mreža

Pri WPF aplikacijah nalogo gradnika `TableLayoutPanel` iz Windows Forms opravlja gradnik `Grid`.

Najprej moramo določiti število vrstic in stolpcev, ki določajo mrežo. V zgledu, ki sledi, smo ustvarili mrežo 4 vrstic s tremi stolpci.

```
<Grid.ColumnDefinitions>
  <ColumnDefinition />
  <ColumnDefinition />
  <ColumnDefinition />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
  <RowDefinition />
  <RowDefinition />
  <RowDefinition />
  <RowDefinition />
</Grid.RowDefinitions>
```

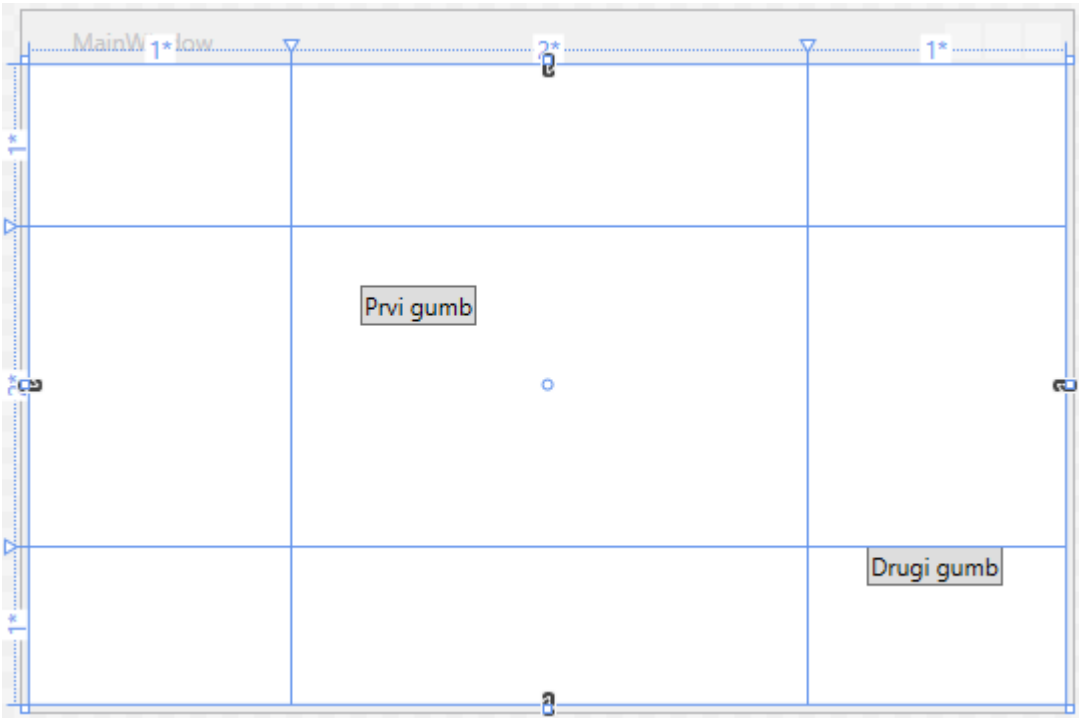
Nastavitev višine vrstic urejamo v lastnosti `RowDefinition.Height` in nastavitev širine v lastnosti `ColumnDefinition.Width`. Privzeto je nastavljeno, da so vse vrstice enako široke in vedno zazamejo vso višino mreže. Podobno velja za stolpce. Lahko pa uporabimo vrednosti `Height` (oz. `Width`), kjer navedemo velikost v pikslih. Namesto pikslov pa lahko s številom zvezdic (*) označimo delež prostora, ki ga zajema vrstica ali stolpec.

Gradnike vstavljamo tako, da njegov opis napišemo znotraj značke Grid. Z lastnostma `Grid.Row` in `Grid.Column` določimo, v katero celico spada. Če želimo vstaviti gradnik tako, da se bo raztezal čez več stolpcev in/ali vrstic, uporabimo lastnosti `Grid.RowSpan` oziroma `Grid.ColumnSpan`.

Lastnosti `HorizontalAlignment` in `VerticalAlignment` opišeta, kje naj bi bil vsebovan gradnik postavljen znotraj celice mreže. Z lastnostmi `Margin` in poravnavami lahko vsakemu gradniku v mreži natančno določimo postavitev.

Več podatkov o gradniku najdete [33].

Poglejmo si primer, kjer na mrežo postavimo dva gumba.



Slika 108: Postavitev gradnikov na mreži

Če pogledamo sliko 108, vidimo, da smo mrežo razdelili na devet celic. Pri tem smo določili višino in širino celic z zvezdno velikostjo. Vidimo, da je širina drugega stolpca dvakrat večja kot preostala dva. Podobno smo nastavili tudi pri vrsticah. Gumb `"Prvi gumb"` smo postavili v sredino prvih dveh stolpcev in vrstic, gumb `"Drugi gumb"` pa smo odložili v celico zadnjega stolpca in vrstice. Pri tem smo nastavili postavitev tega gumba na vrh sredine celice. Za boljše razumevanje odvisnosti vsebovanih gradnikov od celic pa si poglejmo XAML kodo.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="2*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Button Grid.ColumnSpan="2" Grid.RowSpan="2" VerticalAlignment="Center"
     HorizontalAlignment="Center">Prvi gumb</Button>
    <Button Grid.Column="2" Grid.Row="2" HorizontalAlignment="Center"
     VerticalAlignment="Top">Drugi gumb</Button>
</Grid>
```

Najprej smo definirali stolpce in vrstice v mreži ter nastavili njihovo širino oziroma višino. Gumbu `"Prvi gumb"` smo določili, da se raztegne čez dve vrstici in dva stolpca. Ker mu nismo določili lastnosti `Grid.Colum` in `Grid.Row`, smo gumb načeloma postavili v prvo vrsto in prvi stolpec. Ker pa smo uporabili `ColumnSpan` in `RowSpan`, smo ga s tem razširili še na drugo vrstico in drugi stolpec. Ker sta ta vrstica in stolpec dvojne velikosti, dejansko gumb postavljamo na območju, ki ga sestavlja 9 območij tako velikih, kot je npr. celica desno spodaj. Nato smo gumb postavili na sredino tega območja z lastnostma `HorizontalAlignment` in `VerticalAlignment`. Gumbu `"Drugi gumb"` smo nastavili, da je vsebovan v celici zadnje vrstice in zadnjega stolpca. Ker imata prvi stolpec in vrstica indeks 0, vrednost `Grid.Column="2"` in `Grid.Row="2"` predstavljata celico v tretjem stolpcu in vrstici. Z nastavitvijo lastnosti

`VerticalAlignment="Top"` in `HorizontalAlignment="Center"` postavimo gradnik na sredino zgornjega roba te celice.

Label/Oznaka

Tako kot pri Windows Forms je tudi tu osnovni namen tega gradnika prikaz besedila. Ustreznega besedila pa tu ne urejamo v lastnosti `Text`, kot pri Windows Forms, temveč ga urejamo v lastnosti `Content`.

```
<Label Content="Besedilo oznake"/>
```

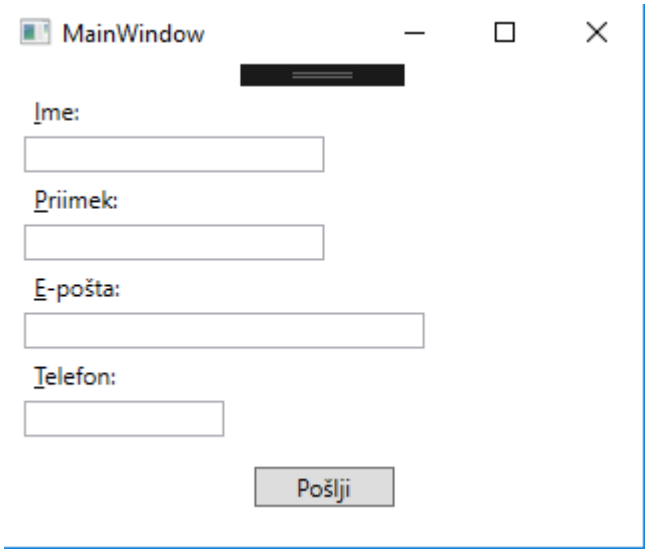
Poleg zapisa besedila v lastnost `Content`, pa lahko besedilo na oznaki prikažemo še tako, da ga uporabimo znotraj značke, torej z:

```
<Label>Besedilo oznake</Label>
```

Kot lahko vidimo, pri tem zapisu nismo dali imena gradniku, kot smo to delali pri Windows Forms aplikacijah, saj to pri WPF aplikacijah ni nujno. Tak način zapisa, da gradniku ne določimo imena (lastnosti `Name`), običajno uporabimo, ko se v kodi ne sklicujemo na ta gradnik.

Če imamo na GUV večje število gradnikov (npr. vnosnih polj), je uporabniku lažje vpisovati podatke tako, da do ustreznega vnosnega mesta pride s kombinacijo tipk ALT + dostopna tipka. Običajno dostopno tipko določimo tako, da v besedilu, ki je prikazano na gradniku, naredimo podčrtaj pred tisti znak, za katerega želimo, da pomeni dostopno tipko. Težava pa nastopi, če bi želeli na ta način dostopati do vnosnega polja. Takrat uporabimo možnost, da z dostopno tipko dostopamo formalno do oznake, ki pa dostop preko lastnosti `Target` prenese na ustrezno vnosno polje.

Na sliki 109 imamo aplikacijo, v kateri želimo, da uporabnik vpiše določene podatke in jih pošlje npr. v podatkovno bazo s klikom na gumb `Pošlji`. Če bi npr. uporabnik pozabil v vnosno polje `E-pošta` vpisati svoj elektronski naslov, bi se moral s pritiskanjem na gumb `TAB` sprehajati med vsemi gradniki znotraj okna, dokler ne bi prišel do želenega vnosnega polja, ali pa bi moral klikniti z miško v pravo vnosno polje.



Slika 109: Ob pritisku na tipko ALT, se nam podčrtajo določene črke gradnika oznaka

V okolju Windows smo navajeni, da za prehajanje med določenimi gradniki uporabimo tipko `ALT` in znak, ki ustreza gradniku, do katerega želimo dostopati. Ob tem vidimo, da se nam ob uporabi tipke TAB podčrtajo znaki, ki smo jih določili kot dostopne tipke. Tako lahko nastavimo, da se npr. s kombinacijo tipk `ALT + E` na tipkovnici, prestavimo v vnosno polje, kamor npr. vpišemo elektronski naslov.

Poglejmo si ustrezno kodo.

```
<Grid>
    <StackPanel Margin="10">
        <Label Target="{Binding ElementName=vpIme}" Content="_Ime:"/>
        <TextBox Name="vpIme" Width="150" HorizontalAlignment="Left"/>
        <Label Target="{Binding ElementName=vpPriimek}" Content="_Priimek:"/>
        <TextBox Name="vpPriimek" Width="150" HorizontalAlignment="Left"/>
        <Label Target="{Binding ElementName=vpEposta}" Content="_E-pošta:"/>
        <TextBox Name="vpEposta" Width="200" HorizontalAlignment="Left"/>
        <Label Target="{Binding ElementName=vpTel}" Content="_Telefon:"/>
        <TextBox Name="vpTel" Width="100" HorizontalAlignment="Left"/>
    </StackPanel>
    <Button Margin="20" VerticalAlignment="Bottom" HorizontalAlignment="Center"
     Width="70">Pošlji</Button>
</Grid>
```

Da določimo tipko za dostop, na napisu na oznaki dodamo podčrtaj pred znakom, ki predstavlja tipko za dostop (npr. `"_Ime:"` v `<Label Target="{Binding ElementName=vpIme}" Content="_Ime:"/>`). Tu tudi povemo, da se ob kombinaciji tipke `ALT + I` (dostopna tipka) uporabnik premakne v vnosno polje z imenom `vpIme`, saj s `Target` prestavimo položaj vnosnega kazalca v ta gradnik. Na podoben način se lahko uporabnik nato sprehaja med ostalimi vnosnimi polji v aplikaciji (`ALT + P`, `E` in `T`).

Ta lastnost je uporabna predvsem, ko imamo na oknu večje število gradnikov, med katerimi se uporabnik sprehaja. Za več informacij o gradniku si poglejte [34].

DataGrid/Podatkovna mreža

Ta gradnik omogoča pregled podatkov v obliki tabele WPF aplikacij, podobno kot to počne gradnik `DataGridView` pri Windows Forms aplikacijah. Za to, da zapolnimo stolpce in vrstice gradnika s podatki, uporabimo lastnost `ItemsSource`. Podatkovna mreža vsebuje lastnost `AutoGenerateColumns`, ki samodejno ustvari stolpec glede na lastnost podatkovnega objekta. Tako se v primeru, da je objekt npr. tipa `string`, ustvari stolpec tipa `TextBox`. Poleg tega pa lahko ustvarimo svoje stolpce. Tedaj moramo lastnost `AutoGenerateColumns` iz privzete nastaviti na `False`. V gradniku z lastnostmi `SelectionMode` in `SelectionUnit` nastavljamo oznako celic na gradniku. Vrednost lastnosti `SelectionMode` lahko nastavimo na `Single` ali `Extended`, kjer določimo, ali je lahko označena ena ali več enot na enkrat. V lastnosti `SelectionUnit` pa določimo, kaj ta enota je. Določimo lahko, da je to ena celica, več celic ali vrstica. V gradniku imamo tudi lastnosti, ki nam pomagajo pri organiziranju podatkov v tabeli. Podobno kot pri gradniku `DataGridView` imamo tudi pri gradniku `DataGrid` lastnosti, s katerimi lahko uporabniku omogočimo urejanje podatkov, ki so prikazani na gradniku. To so lastnosti `CanUserSortColumns`, `CanUserResizeColumns`, `CanUserReorderColumns`. Tako lahko uporabniku omogočimo sortiranje podatkov v stolpcih, spremembo širine stolpcev in spremembo postavitve stolpcev. V lastnosti `HeaderVisibility` pa določimo, kateri naslovi so prikazani. Lahko izbiramo med vrednostmi `None`, `Row`, `Column` ali `All`.

Pri WPF aplikacijah si pomagamo s podatkovno vezavo pri prikazu podatkov iz zunanjih virov. Kot smo spoznali že v razdelku `ListBox`, podatkovna vezava predstavlja povezavo, ki sinhronizira podatke med virom podatkov (npr. bazo podatkov) in uporabniškim vmesnikom aplikacije (v našem primeru gradnik `GridView`). Na primer, ko se spremeni določen podatek v podatkovni bazi, se izpis podatkov na gradniku `GridView` spremeni samodejno. Lahko pa tudi spremenimo zapis v gradniku in se samodejno spremeni zapis v podatkovni bazi. Podatke lahko v aplikaciji vežemo enosmerno ali dvosmerno. Dvosmerna vezava pomeni, da se podatek spremeni, ne glede, kje se je le-ta spreminjal – ali pri viru ali na uporabniškem vmesniku aplikacije. Pri enosmerni vezavi pa lahko podatek spremenimo le v eni smeri.

Na primeru si oglejmo, kako uporabimo podatkovno vezavo pri gradniku DataGrid. Na sliki 110 vidimo, kako v XAML urejevalniku povežemo določene podatke iz vira. V našem primeru je vir podatkov podatkovna baza s tabelo varovanih objektov in dogodki. Na gradniku DataGrid želimo prikazati to tabelo s štirimi stolpci (Številka objekta, Ime naročnika, Dogodek in Datum). Stolpce ustvarimo v DataGridu, ki ga poimenujemo mreza, in jih povežemo s stolpci iz podatkovne baze.

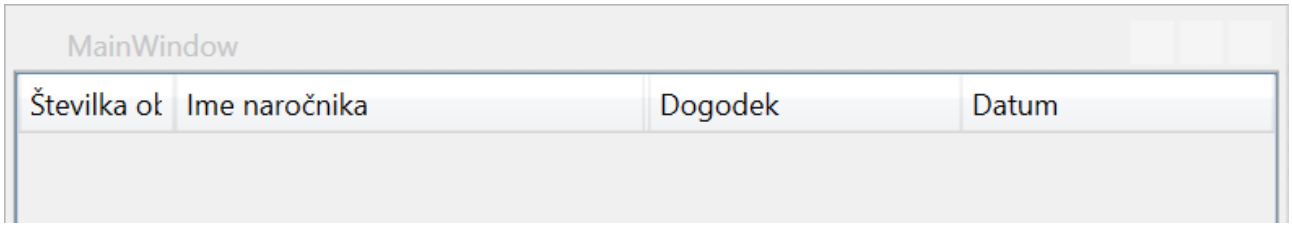
```
SqlConnection povezava = new SqlConnection(@"Data Source=desktop-k78onnb\sqlexpress;Initial
Catalog=PorocilaVNC;Integrated Security=True");

public MainWindow()
{
    InitializeComponent();

    povezava.Open();
    SqlCommand ukaz = new SqlCommand();
    ukaz.CommandText = "SELECT st_objekta, ime_narocnika, dogodek, du_vklop"
    ukaz.CommandText += " FROM center JOIN objekti ON center.ID_objekti = objekti.ID";
    ukaz.CommandText += " WHERE du_zakljucka is null";
    ukaz.Connection = povezava;
    SqlDataAdapter da = new SqlDataAdapter(ukaz);
    DataTable tabelaPodatkov = new DataTable();
    da.Fill(tabelaPodatkov);
    povezava.Close();
    mreza.ItemsSource = tabelaPodatkov.DefaultView;
}
```

Iz zgornjega zapisa vidimo, da v urejevalniku kode zaženemo SQL povpraševanje, ki nam vrne podatke iz podatkovne baze. Ti podatki, ki jih dobimo, so razporejeni po stolpcih. To so st_objekta, ime_naročnika, dogodek in du_vklop, ki predstavlja datum ter uro prihajajočega dogodka. Te podatke nato dodamo v tabelo tabelaPodatkov, ki jo povežemo z gradnikom DataGrid (mreza).

V XAML urejevalniku nato uredimo predstavitev teh podatkov. To naredimo tako, da na gradniku DataGrid ustvarimo štiri stolpce (DataGridTextColumn) in jih povežemo s podatki iz SQL povpraševanja. Tako povežemo prvi stolpec s stolpcem st_objekta iz SQL povpraševanja (Binding="{Binding st_objekta}"), drugi stolpec z ime_narocnika iz SQL povpraševanja (Binding="{Binding ime_narocnika}") in tako še tretji in četrti stolpec. Ker želimo, da uporabniki vidijo tudi ime stolpcev, v katerem so podatki, le-te poimenujemo. To naredimo tako, da vsakemu stolpcu dodamo naslov, ki ga zapišemo v lastnost Header.

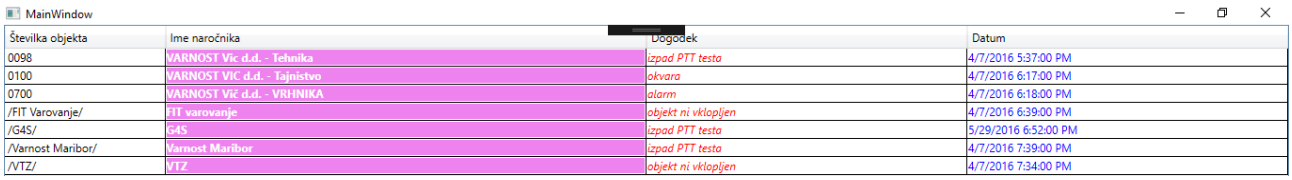
```
<DataGrid Name="mreza" AutoGenerateColumns="False" HeadersVisibility="Column"
    MouseDoubleClick="mreza_MouseDoubleClick" IsReadOnly="True">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding st_objekta}" Header="Številka objekta"
        Width="*"/>
        <DataGridTextColumn FontWeight="Bold" Binding="{Binding ime_narocnika}"
        Header="Ime naročnika" Width="3*">
            <DataGridTextColumn.ElementStyle>
                <Style TargetType="{x:Type TextBlock}">
                    <Setter Property="Foreground" Value="White"></Setter>
                    <Setter Property="Background" Value="Violet"></Setter>
                </Style>
            </DataGridTextColumn.ElementStyle>
        </DataGridTextColumn>
        <DataGridTextColumn FontStyle="Italic" Binding="{Binding dogodek}" Header="Dogodek"
        Width="2*">
        <DataGridTextColumn.ElementStyle>
            <Style TargetType="{x:Type TextBlock}">
                <Setter Property="Foreground" Value="Red"></Setter>
            </Style>
```

```
        </DataGridTextColumn.ElementStyle>
    </DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding du_vklop}" Header="Datum" Width="2*">
            <DataGridTextColumn.ElementStyle>
                <Style TargetType="{x:Type TextBlock}">
                    <Setter Property="Foreground" Value="Blue"></Setter>
                </Style>
            </DataGridTextColumn.ElementStyle>
        </DataGridTextColumn>
    </DataGrid.Columns>
</DataGrid>
```



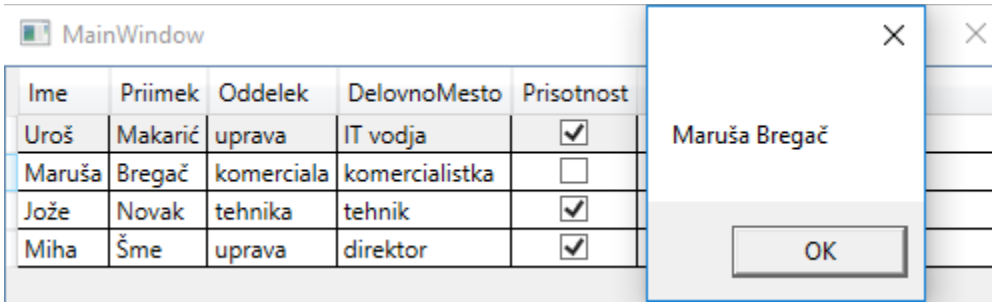
Slika 110: Ustvarjanje stolpcev na gradniku DataGrid v XAML-u

Iz zgornje kode in slike 111 vidimo, kako so podatki iz podatkovne baze prikazani na gradniku `DataGrid` aplikacije. Pri tem smo v XAML urejevalniku vsakem stolpcu gradnika `GridView` spremenili tudi določene lastnosti zapisov. Tako so vsi zapisi v prvem stolpcu zapisani s črno barvo, v drugem s svetlo zeleno, tretjem rdečo in četrtem z modro barvo. Na tak način še bolj odločno ločimo podatke med stolpci.



Slika 111: Podatki iz podatkovne baze, prikazani v WPF aplikaciji

Poglejmo si primer, kjer uporabimo nekatere lastnosti gradnika `GridView`. Tako kot smo pri gradniku `DataGridView` prikazali seznam zaposlenih v podjetju, si poglejmo tudi, kako to naredimo pri gradniku `DataGrid`. Ko uporabnik izbere določeno vrstico, mu v sporočilnem oknu izpišemo vrednosti iz stolpcev z naslovom `Ime` in `Priimek`.



Slika 112: Podatki v gradniku DataGrid

Ker želimo, da se ob kliku na določen podatek v gradniku označi celotna vrstica, tako kot smo to naredili pri gradniku `DataGridView`, tu nastavimo vrednost `SelectionMode` na `Single` in `SelectionUnit` na `FullRow`, saj želimo, da uporabnik izbere na mreži le eno vrstico. Ob tem smo dodali tudi dogodka `Loaded` in `SelectionChanged`.

```
<DataGrid
    x:Name="podatkovnaMreza"
    HorizontalAlignment="Stretch"
```

```
    VerticalAlignment="Stretch"
    SelectionMode="Single"
    SelectionUnit="FullRow"
    Loaded="podatkovnaMreza_Loaded"
    SelectionChanged="podatkovnaMreza_SelectionChanged">
</DataGrid>
```

V kodi smo nastavili, da se ob zagonu aplikacije zažene metoda `podatkovnaMreza_Loaded()`, v kateri dodamo zaposlene v seznam `podatki`, ki jih prikažemo na gradniku `podatkovnaMreza`. V seznam dodamo vsakega zaposlenega posebej z imenom, priimkom, podatkom o oddelku, v katerem dela, na kakšnem delovnem mestu je zaposlen in ali je trenutno v službi ali na dopustu (če je v službi, nastavimo vrednost na `true`).

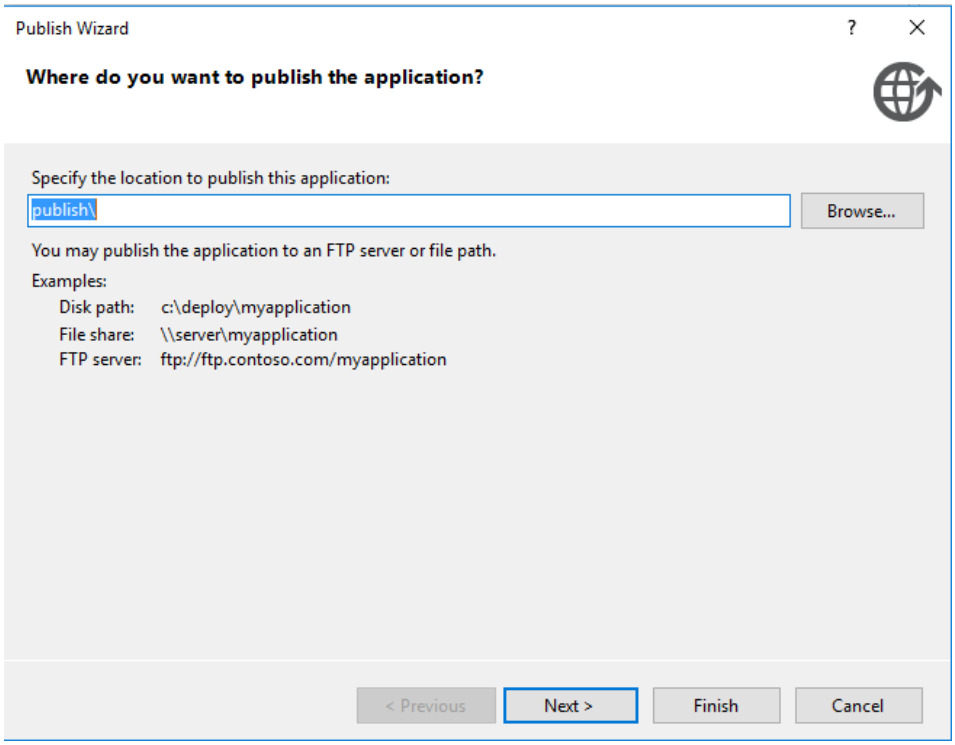
```
public MainWindow()
{
    InitializeComponent();
}

private void podatkovnaMreza_Loaded(object sender, RoutedEventArgs e)
{
    //dodamo nekaj zaposlenih v seznam, ki jih prikažemo na gradniku DataGrid
    var podatki = new List<Zaposleni>();
    podatki.Add(new Zaposleni("Uroš", "Makarić", "uprava", "IT vodja", true));
    podatki.Add(new Zaposleni("Maruša", "Bregač", "komerciala", "komercialistka", false));
    podatki.Add(new Zaposleni("Jože", "Novak", "tehnika", "tehnik", true));
    podatki.Add(new Zaposleni("Miha", "Šme", "uprava", "direktor", true));

    var podatkovnaMreza = sender as DataGrid;
    podatkovnaMreza.ItemsSource = podatki;
}

private void podatkovnaMreza_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    var mreza = sender as DataGrid;
    var izbranaVrstica = mreza.SelectedItems;

    List<string> podatkiZaposlenega = new List<string>();
    foreach (var stolpec in izbranaVrstica)
    {
        var zaposleni = stolpec as Zaposleni;
        podatkiZaposlenega.Add(zaposleni.Ime);
        podatkiZaposlenega.Add(zaposleni.Priimek);
    }
    //izpis imena in priimka izbranega zaposlenega v sporočilnem oknu
    MessageBox.Show(string.Join(" ", podatkiZaposlenega));
}
```

V metodi `podatkovnaMreza_SelectionChanged()`nastavimo, da se ob kliku na celico znotraj gradnika odpre sporočilno okno, v katerem sta zapisana, iz vrstice v kateri je uporabnik kliknil na celico, podatka iz stolpcev `Ime` in `Priimek`. Za več informacij o gradniku si poglejte [35].

Priprava aplikacij za namestitev

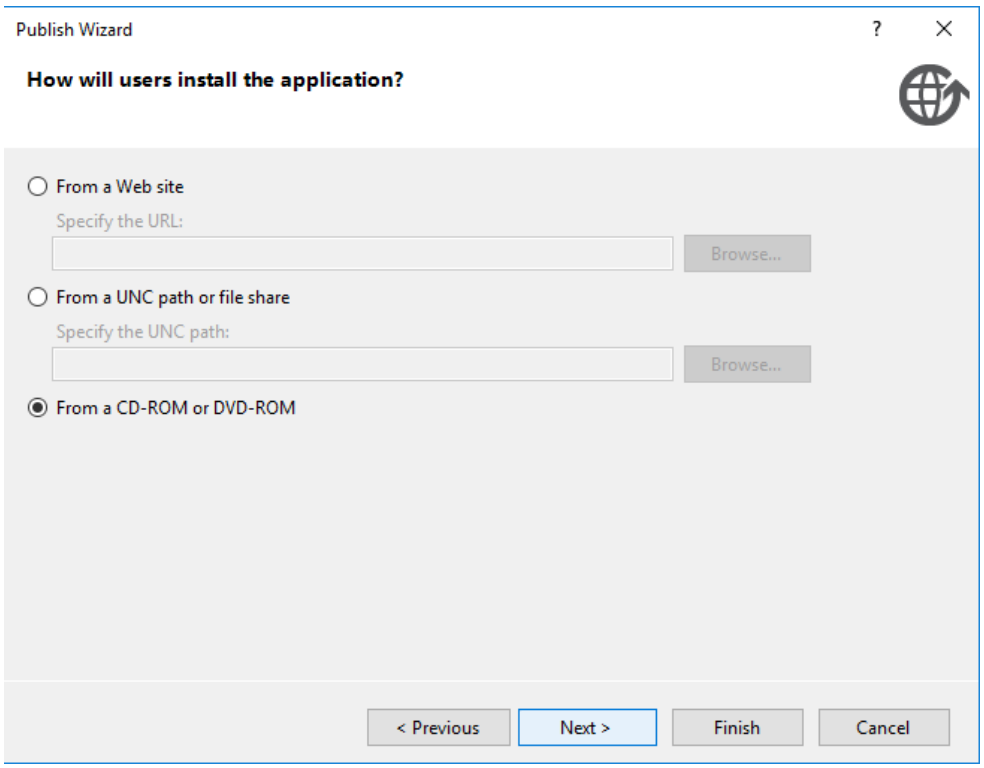
Ko želimo aplikacijo uporabljati na drugih računalnikih, kjer okolja Visual Studio (morda) nimamo nameščenega (ali pa želimo aplikacijo uporabljati izven okolja Visual Studia), moramo ustvariti ustrezno namestitveno datoteko. V ta namen je pripravljen poseben sklop korakov, tako imenovani Namestitveni čarovnik (`Publish Wizard`).

V Solution Explorer izberemo ime projekta in kliknemo nanj z desnim miškinim gumbom. Ponudi se nam izbor možnosti, kjer izberemo `Publish`. S tem zaženemo `Publish Wizard`, kjer nato izberemo mesto, kamor nam bo Visual Studio zapakiral ustrezne datoteke. Z gumbom `Browse ...` poiščemo primerno lokacijo (slika 113).



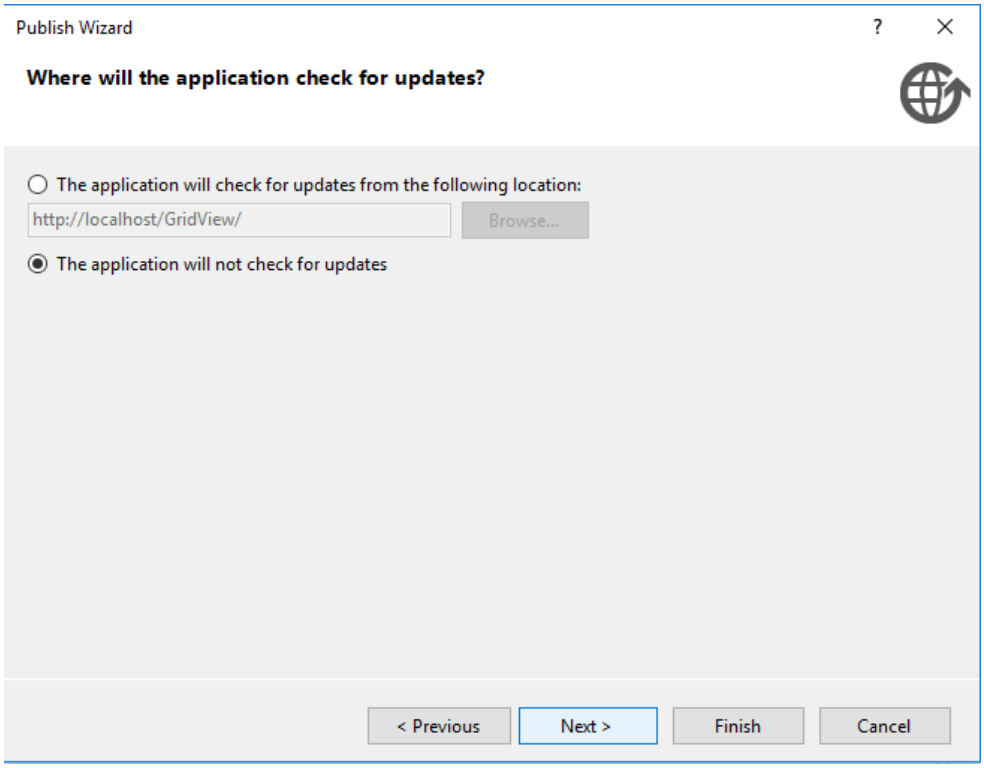
Slika 113: Kje bomo shranili namestitvene datoteke

Na sliki 114 vidimo, da moramo izbrati, kako bodo uporabniki namestili aplikacijo. Na voljo so tri možnosti.



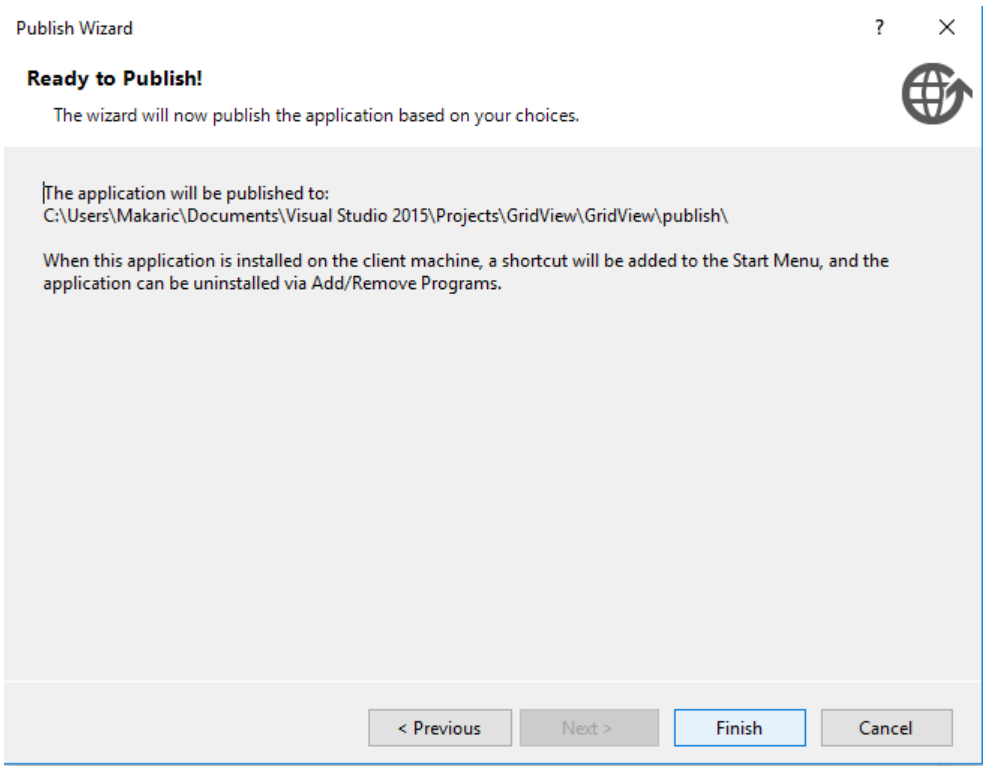
Slika 114: Izbira načina namestitve

Glede na to, da se večina aplikacij razvija, večina razvijalcev ponudi uporabnikom posodobitev aplikacije oz. to, da aplikacija vsake toliko časa preveri, ali je prišlo do kakšnih osvežitev. Te posodobitve se praviloma prenašajo z določene spletne strani. To stran določimo na naslednjem koraku. Če pa svoje aplikacije ne bomo več spreminjali, izberemo nastavitev kot na sliki 115.



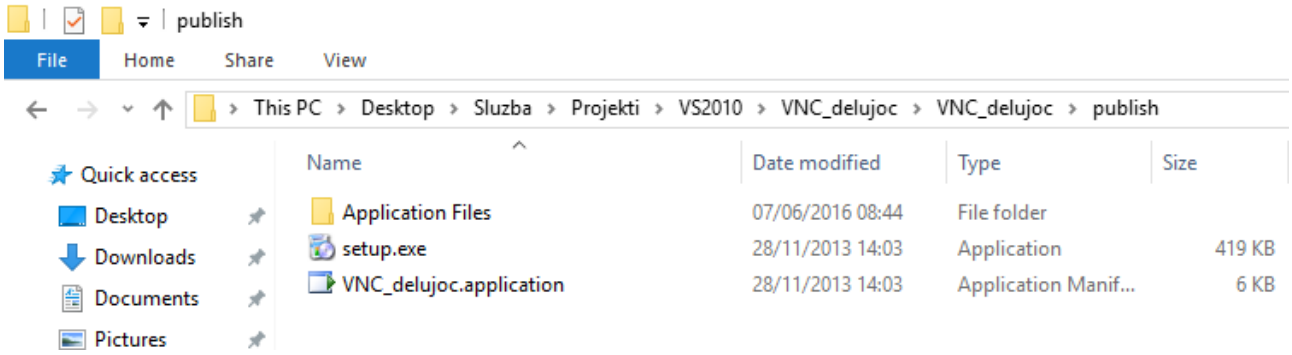
Slika 115: Tretji korak pri izdelavi namestitvene aplikacije

V naslednjem oknu (slika 116) samo še kliknemo na gumb `Finish`, ki ustvari datoteke, potrebne za namestitev aplikacije na prej izbranem mestu oz. naslovu [36].



Slika 116: Zaključni korak pri izdelavi namestitvene aplikacije

Na sliki 117 vidimo datoteke, ki so se ustvarile za namestitev aplikacije. Če želimo namestiti to aplikacijo na drugi računalnik, moramo samo nanj prenesti mapo z vsemi temi datotekami (s pomočjo USB ključka, CD ali DVD …) in zagnati datoteko `setup`.



Slika 117: Primer mape, kjer so shranjene datoteke za zagon programa

Zaključek

Diplomska naloga je razdeljena na dva dela. V prvem smo spoznali razvojno okolje Visual Studio Community in orodja v tem okolju, s katerimi si pomagamo pri razvoju aplikacij. V tem razvojnem okolju lahko razvijamo različne programe in aplikacije. Mi smo podrobneje pogledali tiste prijeme, ki jih uporabimo, ko razvijamo aplikacije z grafičnim uporabniškim vmesnikom, saj so to aplikacije, s katerimi večinoma delamo v vsakdanjem življenju.

V drugem delu diplomske naloge smo si ogledali gradnike, s katerimi razvijamo in ustvarjamo programe, ki imajo grafični uporabniški vmesnik. Predstavili smo dva pristopa, ki ju lahko uporabimo pri razvoju tovrstnih aplikacij, in sicer Windows Forms in WPF. Pri tem smo ugotovili, da imamo pri razvoju WPF aplikacij več možnosti pri oblikovanju grafičnega uporabniškega vmesnika. Ob tem smo spoznali nekatere podobnosti in razlike med gradniki Windows Forms in WPF aplikacij. Ker je število gradnikov veliko, smo si ogledali le nekaj najbolj osnovnih. Izbrali smo tiste, ki jih pogosto uporabljamo pri razvoju. Pri vseh opisanih gradnikih smo razložili le njihove osnovne lastnosti.

Na koncu smo si še ogledali, kako aplikacijo pripravimo za namestitev na računalnik.

Viri in literatura

[1] „Event Driven Programming,“ [Elektronski]. Pridobljeno iz: http://c2.com/cgi/wiki?EventDrivenProgramming. [Dostop 25 01 2016].

[2] „Event-driven programming,“ [Elektronski]. Pridobljeno iz: https://en.wikipedia.org/wiki/Event-driven_programming. [Dostop 06 02 2016].

[3] „Event-driven Program,“ [Elektronski]. Pridobljeno iz: https://www.techopedia.com/definition/7083/event-driven-program. [Dostop 06 02 2016].

[4] „Signing in to Visual Studio,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/dn457348.aspx. [Dostop 2016 02 06].

[5] „Windows Presentation Foundation vs WinForms,“ [Elektronski]. Pridobljeno iz: http://www.infragistics.com/community/blogs/devtoolsguy/archive/2015/04/17/windows-presentation-foundation-vs-winforms.aspx. [Dostop 10 1 2015].

[6] „Windows Presentation Foundation on the Web: Web Browser Applications,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/aa480223.aspx. [Dostop 06 02 2016].

[7] „Style and template overview,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/cc295273.aspx. [Dostop 10 05 2016].

[8] „Styling and Templating,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/ms745683(v=vs.100).aspx. [Dostop 10 05 2016].

[9] „Templates in WPF,“ [Elektronski]. Pridobljeno iz: http://www.c-sharpcorner.com/UploadFile/mahesh/templates-in-wpf/. [Dostop 10 05 2016].

[10] „WPF Control Templates - Creating a Template,“ [Elektronski]. Pridobljeno iz: http://www.blackwasp.co.uk/WPFControlTemplates.aspx. [Dostop 10 05 2016].

[11] „WPF Control Templates – An Overview,“ [Elektronski]. Pridobljeno iz: https://blogs.msdn.microsoft.com/jitghosh/2007/12/27/wpf-control-templates-an-overview/. [Dostop 10 05 2016].

[12] „WPF vs. WinForms,“ [Elektronski]. Pridobljeno iz: http://www.wpf-tutorial.com/about-wpf/wpf-vs-winforms/. [Dostop 10 01 2015].

[13] „Visual C# Code Editor Features,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/9bt0w6h6(v=vs.90).aspx. [Dostop 22 01 2016].

[14] „Intellisense,“ [Elektronski]. Pridobljeno iz: https://www.techopedia.com/definition/24580/intellisense. [Dostop 10 1 2015].

[15] „Visual C# IntelliSense,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/43f44291(v=vs.140).aspx. [Dostop 21 01 2016].

[16] „Using IntelliSense,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx. [Dostop 10 1 2015].

[17] „Debugging,“ [Elektronski]. Pridobljeno iz: http://www.dotnetperls.com/debugging. [Dostop 11 04 2016].

[18] „Debugging in Visual Studio,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/sc65sadd.aspx. [Dostop 11 04 2016].

[19] „Debug your app with Visual Studio,“ [Elektronski]. Pridobljeno iz: https://www.visualstudio.com/en-us/get-started/code/debug-your-app-vs. [Dostop 11 04 2016].

[20] „Debugger,“ [Elektronski]. Pridobljeno iz: https://en.wikipedia.org/wiki/Debugger. [Dostop 10 04 2016].

[21] „Solutions and Projects in Visual Studio,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/b142f8e7.aspx. [Dostop 7 6 2016].

[22] „Solution Explorer,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/bbck0dh6(v=vs.110).aspx. [Dostop 28 04 2016].

[23] „Label Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.label(v=vs.110).aspx. [Dostop 25 02 2016].

[24] „Button,“ [Elektronski]. Pridobljeno iz: http://www.dotnetperls.com/button. [Dostop 29 02 2016].

[25] „CheckBox Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.checkbox(v=vs.110).aspx. [Dostop 10 07 2016].

[26] „RadioButton Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.radiobutton(v=vs.110).aspx. [Dostop 11 07 2016].

[27] „ListBox Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.listbox(v=vs.110).aspx. [Dostop 11 07 2016].

[28] „TextBox Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.textbox(v=vs.110).aspx. [Dostop 13 07 2016].

[29] „TableLayoutPanel Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.tablelayoutpanel(v=vs.110).aspx. [Dostop 14 07 2016].

[30] „Panel Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.panel(v=vs.110).aspx. [Dostop 15 07 2016].

[31] „DataGridView Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.datagridview(v=vs.110).aspx. [Dostop 16 07 2016].

[32] „Timer Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.timers.timer(v=vs.110).aspx. [Dostop 16 07 2016].

[33] „Grid Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.controls.grid(v=vs.110).aspx. [Dostop 17 06 2016].

[34] „Label Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.controls.label(v=vs.110).aspx. [Dostop 22 07 2016].

[35] „DataGrid Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.controls.datagrid(v=vs.110).aspx. [Dostop 17 07 2016].

[36] „How to: Publish a ClickOnce Application using the Publish Wizard,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/31kztyey.aspx. [Dostop 29 02 2016].

[37] Programming WPF: Building Windows UI with Windows Presentation Foundation, California: O'Reilly, 2007.

[38] „Model–view–viewmodel,“ [Elektronski]. Pridobljeno iz: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel. [Dostop 02 04 2016].

[39] „WPF tutorial,“ [Elektronski]. Pridobljeno iz: http://www.wpf-tutorial.com/data-binding/introduction/. [Dostop 02 04 2016].

[40] „WPF Tutorial.net,“ [Elektronski]. Pridobljeno iz: http://www.wpftutorial.net/DataBindingOverview.html. [Dostop 02 04 2016].

[41] „Introducing WPF – Experiences of a former Windows Forms developer,“ [Elektronski]. Pridobljeno iz: http://www.centigrade.de/blog/en/article/introducing-wpf-experiences-of-a-former-windows-forms-developer/. [Dostop 07 04 2016].

[42] „Data Binding,“ [Elektronski]. Pridobljeno iz: https://www.techopedia.com/definition/15652/data-binding. [Dostop 06 04 2016].

[43] „Navigating through Code with the Debugger,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/y740d9d3.aspx. [Dostop 11 04 2016].

[44] „Visual C# .NET,“ [Elektronski]. Pridobljeno iz:

http://uranic.tsckr.si/VISUAL%20C%23/VISUAL%20C%23.pdf. [Dostop 05 04 2016].

[45] „5 Awesome Visual Studio Debugger Features,“ [Elektronski]. Pridobljeno iz: http://www.cprogramming.com/tutorial/visual_studio_tips.html. [Dostop 05 04 2016].

[46] „Parameter Info,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/25cey46e(v=vs.100).aspx#. [Dostop 03 04 2016].

[47] „Refactoring,“ [Elektronski]. Pridobljeno iz: http://www.dreamincode.net/forums/topic/77242-refactoring/. [Dostop 27 04 2016].

[48] „Refactoring (C#),“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/719exd8s.aspx. [Dostop 27 04 2016].

[49] „Extract Method,“ [Elektronski]. Pridobljeno iz: https://sourcemaking.com/refactoring/extract-method. [Dostop 27 04 2016].

[50] „WPF: How to Template, Style, and Animate a WPF Button Control,“ [Elektronski]. Pridobljeno iz: https://www.youtube.com/watch?v=W8GoWHSkT4g. [Dostop 10 05 2016].

[51] „Using WPF styles,“ [Elektronski]. Pridobljeno iz: http://www.wpf-tutorial.com/styles/using-styles/. [Dostop 10 05 2016].

[52] „Introduction to WPF Templates,“ [Elektronski]. Pridobljeno iz: http://www.codeproject.com/Articles/30994/Introduction-to-WPF-Templates. [Dostop 10 05 2016].

[53] „Control Templates,“ [Elektronski]. Pridobljeno iz: http://www.wpftutorial.net/templates.html. [Dostop 10 05 2016].

[54] „Mastering Visual Studio .NET,“ [Elektronski]. Pridobljeno iz: https://www.safaribooksonline.com/library/view/mastering-visual-studio/0596003609/ch01.html. [Dostop 08 06 2016].

[55] „ErrorProvider Class,“ [Elektronski]. Pridobljeno iz: https://msdn.microsoft.com/en-us/library/system.windows.forms.errorprovider(v=vs.110).aspx. [Dostop 15 07 2016].

[56] J. V. H. J. D. R. Benjamin Perkins, Beginning C# 6 Programming with Visual Studio 2015, Wrox, 2015.

[57] R. Whitaker, The C# Player's Guide (2nd Edition), Starbound Software, 2015.

[58] J. Sharp, Microsoft Visual C# Step by Step (8th Edition) (Developer Reference), Microsoft Press, 2015.