Univerza v Mariboru
Fakulteta za elektrotehniko,
računalništvo in informatiko

Matej Spindler

RAZVOJ NAMIZNIH APLIKACIJ V WPF Z ARHITEKTURNIM VZORCEM MVVM IN OGRODJEM MEF

Diplomsko delo

Ptuj, maj 2011

Diplomsko delo visokošolskega strokovnega študijskega programa

RAZVOJ NAMIZNIH APLIKACIJ V WPF Z ARHITEKTURNIM VZORCEM MVVM IN OGRODJEM MEF

Študent: Matej Spindler

Študijski program: VS ŠP Informatika in tehnologije komuniciranja

Smer: Razvoj informacijskih sistemov

Mentor(ica): viš. pred. mag. Boštjan Kežmah, dipl. gos. inž. elektroteh.

Ptuj, maj 2011

Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko

Številka: BITK.33
Datum in kraj: 16. 06. 2011, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 1/2010)

SKLEP O DIPLOMSKEM DELU

1. **Mateju Spindlerju,** študentu visokošolskega strokovnega študijskega programa INFORMATIKA IN TEHNOLOGIJE KOMUNICIRANJA, smer Razvoj informacijskih sistemov, se dovoljuje izdelati diplomsko delo pri predmetu Načrtovalski vzorci.

2. **MENTOR: viš. pred. mag. Boštjan Kežmah**

3. **Naslov diplomskega dela:**
   RAZVOJ NAMIZNIH APLIKACIJ V WPF Z ARHITEKTURNIM VZORCEM MVVM IN OGRODJEM MEF

4. **Naslov diplomskega dela v angleškem jeziku:**
   WPF DESKTOP APPLICATIONS DEVELOPMENT USING ARCHITECTURAL PATTERN MVVM AND MEF FRAMEWORK

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (dva trdo vezana izvoda in en v spiralo vezan izvod) ter en izvod elektronske verzije do 16. 06. 2012 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:

UNIVERZA V MARIBORU FAKULTETA ZA ELEKTROTEHNIKO, RAČUNALNIŠTVO IN INFORMATIKO 1

Obvestiti:
- kandidata,
- mentorja,
- odložiti v arhiv.

ZAHVALA

Zahvaljujem se mentorju viš. pred. mag. Boštjanu Kežmahu za pomoč in vodenje pri opravljanju diplomske naloge. Zahvaljujem se tudi staršema, ki sta mi omogočila študij.

RAZVOJ NAMIZNIH APLIKACIJ V WPF Z ARHITEKTURNIM VZORCEM MVVM IN OGRODJEM MEF

Ključne besede: MVVM, WPF, MEF, C#, načrtovalski vzorci, namizne aplikacije

UDK: 004.439(043.2)

Povzetek

V diplomskem delu je predstavljena tehnologija Windows Presentation Foundation, ki je namenjena razvoju aplikacij z bogatim uporabniškim vmesnikom.

Za boljši in lažji razvoj pa bomo spoznali tudi arhitekturen vzorec Model-Pogled-Model Pogleda (ang. Model-View-View Model) in ogrodje za upravljanje razširitev (ang. Managed Extensibility Framework).

DESKTOP APPLICATION DEVELOPMENT IN WPF WITH ARCHITECTURAL PATTERN MVVM AND MEF

Key words: MVVM, WPF, MEF, C#, design patterns, desktop applications

UDK: 004.439(043.2)

Abstract

In this diploma work we introduced Windows Presentation Foundation, which is intended for desktop application development with rich user interface.

For a better and easier development, we will present an architectural pattern Model-View-View model and a framework for managing extensibility, called Managed Extensibility Framework.

VSEBINA

Uporabljene kratice

WPF – Windows Presentation Foundation

MVVM – Model-View-View Model

MEF – Managed Extensibility Framework

XAML – Extensible Application Markup Language

XBAP – Xaml Browser Application

CSS – Cascading Style Sheets

HTML – Hyper Text Markup Language

MVP – Model-View-Presenter

MVC – Model-View-Controller

PM – Presentation Model

ASP.NET – Active Server Pages

UI – User Interface

1. UVOD

Gradnja profesionalnih namiznih aplikacij ni lahka naloga, predvsem, če želimo, da je naša programska oprema robustna, razširljiva, uporabniku prijazna ter testabilna.

Za sam razvoj aplikacij imamo na voljo kar nekaj različnih tehnologij, ki nam omogočajo različne stvari. Ena izmed njih je Microsoft Windows Presentation Foundation (WPF), ki jo bomo obravnavali v prvem delu diplomske naloge. WPF je platforma naslednje generacije za razvoj uporabniško bogatih vmesnikov in je del Microsoftovega ogrodja .NET od verzije 3.0 naprej. Uporabniki vedno želimo bogat, interaktiven in vizualno privlačen uporabniški vmesnik; prav s tega vidika je WPF zelo fleksibilen. Z bogatim uporabniškim vmesnikom pa zadovoljimo le uporabnike.

Aplikacije običajno zahtevajo spreminjanje funkcionalnosti in vzdrževanja, kar pa je lahko oteženo v primeru slabe implementacije. V pomoč so nam različni načrtovalski vzorci.

Eden izmed teh vzorcev je arhitekturen vzorec MVVM (Model-View-View Model), ki ga bomo obravnavali v drugem poglavju.

MVVM je primeren predvsem za aplikacije, ki so grajene v WPF ali pa Silverlight. Razvoj aplikacije po tem vzorcu ima mnogo prednosti, predvsem neodvisnost gradnje uporabniškega vmesnika od implementacije logike in omogočeno testiranje. Običajno pa pride pri programski opremi tudi do razširitev oz. nadgradenj, kar je lahko velika težava, zlasti v primeru slabe implementacije ter načrtovanja. Zadnje čase je v programski opremi tudi velik poudarek na vtičnikih (ang. Plugins). Pri teh težavah nam pomaga ogrodje za upravljanje razširljivosti (ang. Managed Extensibility Framework), ki nam omogoča uvoz novih funkcionalnosti; podrobneje ga bomo opisali v zadnjem poglavju diplomskega dela.

Z vsem tem bomo izpolnili namen diplomskega dela, ki je spoznati način gradnje dobrih namiznih aplikacij v WPF.

Skozi diplomsko delo so predstavljeni tudi posamezni primeri, in sicer v jezikih XAML za vmesnik ter v C# za proceduralno kodo.

Kot omejitev diplomske naloge je treba izpostaviti predvsem praktičen del, kjer ni predstavljena celotna aplikacija, temveč le del.

2. WINDOWS PRESENTATION FOUNDATION

WPF je Microsoftovo najnovejše ogrodje za izdelavo bogatih uporabniških vmesnikov in je kombinacija uporabniškega vmesnika, 2d-grafike, 3d-grafike, dokumentov ter večpravnostnih vsebin v enem samem ogrodju. V .NET-ogrodju je prisoten od verzije 3.0 naprej. Skozi evolucijo .NET-a je tudi WPF doživel svoje spremembe.

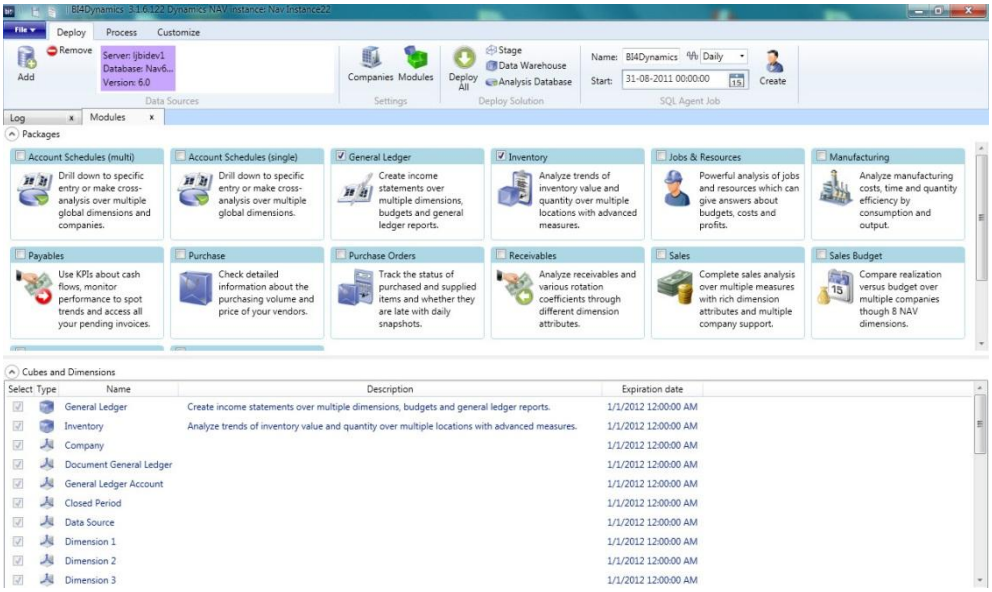
Za sam izris komponent skrbi vektorsko s pomočjo grafične kartice. Tehnično gledano nam WPF ne omogoča ničesar novega. Vprašanje je samo, koliko časa in denarja imamo na voljo za razvoj aplikacij. Če bi želeli ustvariti uporabniški vmesnik ekvivalenten WPF, bi se morali naučiti nove tehnologije (npr. DirectX, OpenGL), prav tako bi morali poskrbeti za pravilen izris pikslov in interakcijo z vhodnimi napravami. Tako bi lahko rekli, da nam WPF pravzaprav prinese nekaj novega, kadar še upoštevamo čas in denar.

DirectX je priročnejši kot WPF za bolj napredne razvijalce, ki pišejo igre ali aplikacije s kompleksnimi 3D-modeli, kjer se potrebuje maksimalna zmogljivost. Kljub temu v DirectX lahko z lahkoto napišemo aplikacijo, ki je po zmogljivosti počasnejša od enake WPF-aplikacije. [12]

WPF nam omogoča gradnjo aplikacije s programsko kodo kot z označevalnim jezikom, podobno, kot je to storjeno v ASP.NET.

Označevalen jezik je imenovan XAML (Extensible Application Markup Language), v katerem se običajno piše uporabniški vmesnik, medtem ko v programskem jeziku pišemo obnašanje. [9]

Največjo prednost pred drugimi tehnologijami ima WPF prav pri izdelavi uporabniškega vmesnika. Pogosto je največji izziv narediti uporaben uporabniški vmesnik, ki je tudi inovativen in privlačnega izgleda, kar je izjemno težko. Prav tukaj nam WPF delo nekoliko olajša.



Slika 1: Aplikacija narejena v WPF

2.1. XAML

V WPF in Silverlight je XAML primarno uporabljen za opisovanje uporabniških vmesnikov, vendar je veliko več kot le-to.

Cilj XAML-a je, da je za razvijalce enostavno delati z eksperti drugih področij, saj je XAML v tem primeru njihov skupen jezik. V WPF so ti eksperti grafični oblikovalci, ki lahko uporabljajo orodja, kot je Expression Blend, za kreacijo uporabniških vmesnikov, medtem ko programerji neodvisno pišejo programsko kodo.

V XAML lahko prav tako kreiramo objekte, kot jih v programski kodi; torej napišemo WPF-aplikacijo tudi popolnoma brez XAML-a, toda takšno početje ni priporočljivo, saj ne izkoristimo prej opisanih prednosti. Je pa nekaj funkcionalnosti XAML-a, ki jih ni mogoče izkoristiti iz proceduralne kode, vendar nas le-te pomanjkljivosti toliko ne omejujejo. [12]

```
<Window x:Class="LogicalVisualTree.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Border CornerRadius="6" BorderBrush="AliceBlue" BorderThickness="3"/>
        <Grid HorizontalAlignment="Center" VerticalAlignment="Center" Height="120" Width="250">
            <Grid.Resources...>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <TextBlock Text="Uporabniško ime:" Grid.Row="0" Grid.Column="0"/>
            <TextBlock Text="Geslo:" Grid.Row="1" Grid.Column="0"/>
            <TextBox Grid.Row="0" Grid.Column="1" />
            <TextBox Grid.Row="1" Grid.Column="1" />
            <Button Content="Prijava" Grid.Row="2" Padding="10,3" Grid.ColumnSpan="2"/>
        </Grid>
    </Grid>
</Window>
```

Slika 2: primer XAML-datoteke

2.1.1. IMENSKI PROSTORI

Imenski prostori so navedeni v korenskem elementu posameznega XAML-dokumenta. Običajno jim dodajamo skupaj predpono, ki jo izberemo po želji.

Največja razlika med XAML in proceduralno kodo se kaže pri imenskih prostorih, saj le-ti niso enaki tako kot pri elementih. Imenski prostori se prevedejo malce drugače, kajti imenski prostor xmlns:x=«http://schemas.microsoft.com/winfx/2006/xaml/presentation« WPF se npr. poveže z več .NET-imenskimi prostori. [12][9]

XAML in proceduralna koda

WPF-aplikacije lahko v celoti napišemo v proceduralni kodi, preproste pa napišemo tudi samo v XAML. WPF ima možnost branja in pisanja XAML-datotek s proceduralne kode; to storimo s pomočjo razredov XamlReader in XamlWriter. Ob pomoči XamlReaderja naložimo želeno XAML-datoteko v objekt ter iz le-tega beremo posamezne elemente. Elementu lahko potem npr. določimo kakšno novo lastnost.

V sami XAML-datoteki je tudi možno pisati proceduralno kodo, in sicer z x:Code elementom in CDATA, vendar za to početje ni dobrega razloga. [10]

Ločitev videza in obnašanja

WPF loči uporabniški vmesnik od svojega obnašanja. Izgled uporabniškega vmesnika je običajno določen v XAML-datoteki, obnašanje pa v .NET-jeziku kot C# ali Visual Basic.

Ta dva dela sta v MVVM-modelu povezana z vezanjem podatkov, ukazov in dogodkov.

Takšna ureditev nam prinaša naslednje ugodnosti:

- Zmanjšani stroški razvoja in vzdrževanja, ker sta grafični vmesnik in sama koda »šibko« povezana.
- Grafični oblikovalci in programerji delajo ločeno, na različnih modelih.
- Uporablja se več različnih orodij za implementacijo XAML in programske kode.
- Lokalizacija in globalizacija sta poenostavljeni. [9]

2.2. Osnove WPF

V nadaljevanju bomo pregledali nekaj glavnih konceptov WPF-a, brez katerih bolj napreden razvijalec skoraj ne more. Nekateri koncepti so novi (npr. vizualno in logično drevo), drugi pa so le nadgradnja že poznanih.

Vsakemu razvijalcu so poznane lastnosti in dogodki, ki so glaven del .NET-a. WPF torej nekako le nadgradi že poznane koncepte.

2.2.1. Odvisnostne lastnosti

Pri razvoju WPF-aplikacij slej ko prej srečamo odvisnostne lastnosti. Na prvi pogled le-te izgledajo podobne običajnim .NET-lastnostim.

Odvisnostne lastnosti so kompleksnejše, in tako tudi uporabnejše. Kadar razumemo njihov temeljni koncept, nam WPF postane veliko bolj intuitiven.

Velika večina lastnosti, ki jih nastavljamo v XAML, so odvisnostne lastnosti, vendar tega na prvi pogled niti ne opazimo. Seveda pa lahko ustvarimo tudi svoje odvisnostne lastnosti, ki jih lahko potem uporabljamo na posameznih kontrolah.

Za osnovno gradnjo WPF-aplikacij teh lastnosti običajno ne ustvarjamo sami, vendar so zelo uporabne, predvsem v navezi z MVVM-vzorcem.

Glede na .NET-lastnosti se odvisnostne lastnosti razlikujejo po tem, da se le-te ustvarijo dinamično, kadar jim priredimo vrednost. Kadar nastavimo vrednost odvisnostne lastnosti, se le-ta ne shrani v spremenljivko, temveč se shrani v slovar ključev in vrednosti. [10][12]

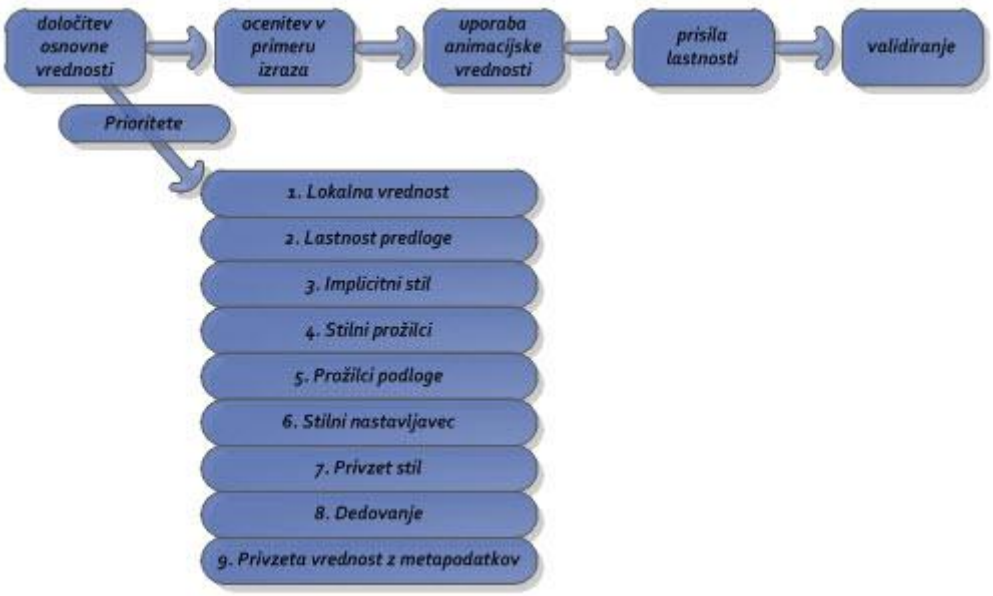
Tri glavne prednosti odvisnostnih lastnosti so:

- Obvestilo sprememb
    - Kadarkoli se vrednost odvisnostne lastnosti spremeni, lahko WPF avtomatsko izvede kakšne akcije. Te so lahko osvežitev uporabniškega vmesnika, sprememba vrednosti na kakšni drugi kontroli, osvežitev vezav podatkov.
- Vrednostno dedovanje
    - Vrednostno dedovanje lastnosti se ne nanaša na tradicionalno dedovanje, ki ga poznamo iz objektno orientiranega programiranja. Odvisnostne spremenljivke dedujejo hierarhično, z zgornjega nivoja navzdol. Če definiramo npr. velikost pisave na prvem nivoju, se le-ta sprememba odraža na spodnjih nivojih. Spremembe pa se vedno ne dedujejo, za kar sta lahko dva razloga:
        - Vse lastnosti ne sodelujejo v dedovanju.
        - Viri z večjo prioriteto nastavijo vrednost. Običajno so to sistemski viri.

- Zmanjšana uporaba delovnega spomina
    - Vzemimo za primer gumb, ki ima preko 130 lastnosti; večina teh običajno ostane na privzetih vrednostih. Brez odvisnostnih lastnosti bi morali hraniti vse lastnosti oz. spremenljivke, čeprav jih pravzaprav ne uporabljamo. Odvisnostne lastnosti ta problem rešijo tako, da so shranjene le tiste, ki so v uporabi. [10][12]

WPF ima posebno strategijo ovrednotenja odvisnostne lastnosti, tako je možno, da je vrednost, ki jo dobimo iz lastnosti, drugačna od pričakovane. Dobljena vrednost lahko izvira iz enega izmed mnogih možnih virov, ki so del WPF-sistema za lastnosti.

Koraki in prioritete pridobivanja vrednosti so naslednji:



Slika 3: Koraki pridobivanja vrednosti v odvisnostni lastnosti [10][12]

Kot vidimo, ima lokalna vrednost, torej vrednost, ki smo jo priredili, najvišjo prioriteto, vendar le pogojno, saj se še lahko ta vrednost spremeni, če uporabljamo kakšno animacijo (npr. ProgressBar) ali pa pri prisili lastnosti. Primer prisile lastnosti bi bil pri uporabi ProgressBar kontrole, če je podana vrednost minimum pod nastavljeno minimalno

vrednostjo, v tem primeru se vrednost prisilno nastavi na minimum. V primeru, da nobena vrednost ni nastavljena, se na koncu vzame privzeta vrednost iz metapodatkov.

PRIMER:

Prikazuje uporabo odvisnostne lastnosti na RichTextbox kontroli, ki je sposobna prikazovati le objekte tipa FlowDocument, mi pa smo potrebovali, da prejme kot vrednost tekst in ga dinamično pretvori v FlowDocument.

```
public static readonly DependencyProperty DocumentProperty = DependencyProperty.Register("TextDocument",
     typeof(String),
     typeof(BindableRichTextBox),
     new FrameworkPropertyMetadata("Ni podatka",
     new PropertyChangedCallback(OnDocumentChanged),
     new CoerceValueCallback(CoerceDocument)));

public new String TextDocument
{
    get
    {
        return (String)this.GetValue(DocumentProperty);
    }

    set
    {
        this.SetValue(DocumentProperty, value);
    }
}

public static void OnDocumentChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args)
{
    // Logika, ki se izvede v primeru, da se dokument spremeni (pretvorba      teksta v FlowDocument)
}

public static object CoerceDocument(DependencyObject obj, object value)
{
    // Logika, ki se izvede nazadnje
    //(preveri npr. pravilnost ustvarjenega FlowDocumenta in če le
    //ta ne ustreza vrne kakšen privzet dokument)
    return _privzetFlowDocument;
}
```

Slika 4: Primer odvisnostne lastnosti

V tem primeru smo torej ustvarili odvisnostno lastnost z imenom TextDocument, ki prejme vrednost tipa String, lastnik je tipa BindableRichTextBox (naš razred, ki deduje od RichTextBox), ima privzeto vrednost »ni podatka«, kadar se lastnost spremeni, se proži metoda OnDocumentChanged in na koncu se preveri pravilnost generiranega dokumenta v metodi CoerceDocument. Uporabili smo torej vse korake za določanje vrednosti odvisnostne lastnosti. Vse seveda niso obvezne, ampak so v tem primeru prišle prav.

Priložene lastnosti

Priložene lastnosti (ang. Attached properties) so posebna vrsta odvisnostnih lastnosti in so neke vrste globalne lastnosti. Obstajajo, da lahko podelementi shranjujejo vrednosti, povezane z nadelementom.

Ena izmed najbolj uporabljenih priloženih lastnosti je prav gotovo DockPanel.Dock, s katero pozicioniramo podelemente odlagalne plošče. [10]

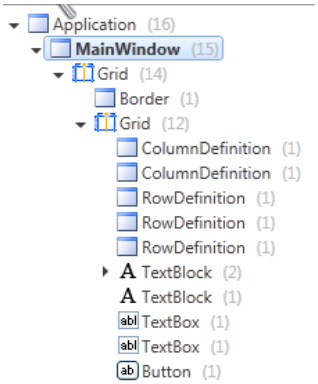
```
<DockPanel>
    <Grid DockPanel.Dock="Left">
        <!--...-->
```

Slika 5: Uporaba priložene lastnosti

2.2.2. Logično in vizualno drevo

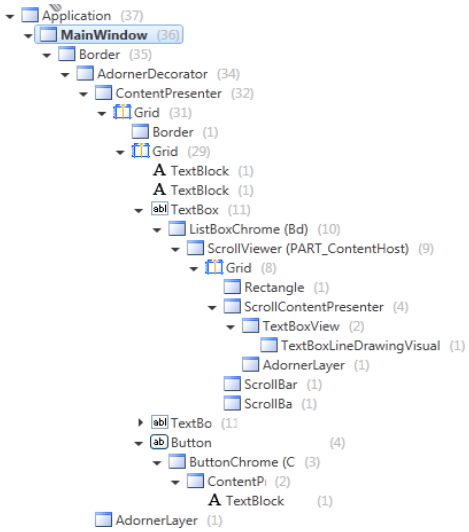
WPF-hierarhična struktura ima nov konceptualen model aplikacijske strukture, ki se predstavi v obliki drevesa. Za ustrezen prikaz strukture potrebujemo dve vrsti dreves, in sicer vizualno ter logično drevo. [13]

Logično drevo je lažje razumljivo, lahko si ga predstavljamo že s pogledom XAML-elementov. Kot je razvidno iz slike 2 in slike 6, sta struktura XAML in logično drevo enaka.



Slika 6: Logično drevo

Vizualna drevesa si težje predstavljamo samo s pogledom XAML-a, saj vsebujejo tudi vsak objekt tipa Visual in Visual3d znotraj posamezne kontrole, ki jih mi pravzaprav ne vidimo v obliki običajne kontrole. [12][13] Na sliki 7 vidimo, da ima preprosta kontrola, kot je TextBox, znotraj gnezdenih kar nekaj kontrol oz. elementov. Kadar za posamezno kontrolo naredimo podlogo, s tem pravzaprav ponovno definiramo vizualno drevo.



Slika 7: Vizualno drevo

WPF nam za delo z drevesi pomaga z razredoma VisualTreeHelper in LogicalTreeHelper, s katerima se lahko npr. sprehajamo skozi drevo ter imamo dostop do posameznih elementov.

2.2.3. Preusmerjeni dogodki

Preusmerjeni dogodki so dogodki, ki krmarijo gor ali dol po vizualnem drevesu glede na svojo preusmerjevalno strategijo (RoutingStrategy).

V primeru, da dogodek obravnavamo, kjer smo ga tudi sprožili, je njegovo obnašanje v veliki meri nevidno. Uporaben postane predvsem, kadar želimo definirati svojo kontrolo ali če definiramo skupen upravljavec dogodka na skupnem korenskem elementu. Dogodki se lahko uporabljajo tudi za komunikacijo skozi kontrolno drevo, kjer podatke spreminjamo v posameznih metodah, kjer se dogodek obravnava. Šele, ko je dogodek označen za obravnavanega s strani upravljavca, le-ta neha s proženjem oz. preusmerjanjem.

Preusmerjevalna strategija je lahko treh vrst:

- Tunneling: Dogodek je najprej sprožen na korenskem nivoju in potem na vseh podelementih do izvirnega elementa oz. dokler ni dogodek označen kot obravnavan s strani upravljavca.
- Bubbling: Dogodek je najprej sprožen na izvirnem elementu in potem gre gor po vizualnem drevesu oz. dokler ne doseže korenskega elementa.
- Direct: Dogodek je prožen le na izvornem elementu

Vzemimo za primer gumb, čigar vizualno drevo je prikazano na sliki 7, s katere je razvidno, da je le-ta sestavljen iz Chrome in TextBlock kontrol. Ob kliku na gumb pravzaprav kliknemo na eno izmed teh dveh kontrol, nato ta dogodek potuje navzgor po vizualnem drevesu, dokler ne pride do gumba, kjer ga obravnavamo. [10][12]

PRIMER DEFINICIJE SVOJEGA PREUSMERJEVALNEGA DOGODKA:

```
//registracija dogodka z imenom MyEvent
public static readonly RoutedEvent MyEvent =
    EventManager.RegisterRoutedEvent("CustomEvent", RoutingStrategy.Bubble,
    typeof(RoutedEventHandler), typeof(MyNewButton));

// .NET ovojnica
public event RoutedEventHandler CustomEvent
{
    add { AddHandler(MyEvent, value); }
    remove { RemoveHandler(MyEvent, value); }
}

// prožilec dogodka
void RaiseMyCustomEvent()
{
    RoutedEventArgs newEventArgs = new RoutedEventArgs(MyNewButton.MyEvent);
    RaiseEvent(newEventArgs);
}

//prepis click metode
protected override void OnClick()
{
    RaiseMyCustomEvent();
}
```

Slika 8: Po meri narejen dogodek

Lastno narejen dogodek prožimo enako kot že obstoječe dogodke.

2.3. Elementi in atributi

XAML-specifikacija definira pravila, ki pretvorijo .NET-imenske prostore, tipe, lastnosti in dogodke v XML-imenske prostore, elemente in atribute.

Med izvajanjem se vedno najprej pripnejo oz. inicializirajo dogodki, nato pa lastnosti, zato sam vrsti red atributov v XAML ni tako pomemben. Več deklariranih lastnosti pa se inicializira tako, kot so navedeni po vrsti v XAML. Zato je potrebno paziti pri navedbi virov (ang. resources), saj vir ne more uporabljati drugega vira, ko le-ta še ne obstaja oz. ni deklariran pred njim. [6][12]

Posamezen element (objekt) ima lahko tri tipe podelementov:

- vrednost lastnosti,

- polje,

- vrednost, ki je pretvorjena v pripadajoč tip.

2.3.1. Lastnosti elementov

V XAML nismo omejeni le na atribute in tekst znotraj elementa, posamezen element ima lahko tudi drugo elemente, ki so lastnosti elementa oz. kontrole, ki jo ustvarjamo.

Tako lahko določimo tudi lastnosti, ki ne prejmejo kot vrednost samo preprost tekst. [10][12]

```
<Button Content="Klik!" Width="120" Background="Blue"/>
```

Slika 9: Definicija gumba z atributi

```
<Button>
    <Button.Content>
        Klik!
    </Button.Content>
    <Button.Width>
        120
    </Button.Width>
    <Button.Background>
        <SolidColorBrush Color="Blue"/>
    </Button.Background>
</Button>
```

Slika 10: Definicija gumba z elementi

2.3.2. Razširitve

Razširitve nam, tako kot pretvorniki tipov, omogočajo, da razširimo izraznost XAML-a. Razširitev XAML-a naredimo takole, da damo izraz v {izraz}, tako XAML-u povemo, da naj izraz jemlje kot razširitev, in ne kot naveden tekst. Posamezne lastnosti razširitve ločimo z vejico.

Razširitev je več vrst, najbolj razširjene razširitve so:

- StaticResource
- DynamicResource
- Binding
- RelativeSource
- TemplateBinding

Vse te razširitve bomo obravnavali v nadaljnjih poglavjih. [9][12]

```
<Button Content="Klik!" Style="{StaticResource BlueButtonStyle}"/>
```

Slika 11: Primer razširitve

2.4. Postavitev aplikacije

Postavitev aplikacije je verjetno ena izmed najbolj pomembnih stvari pri gradnji namiznih aplikacij, vsaj z uporabniškega vidika.

Slabo postavljeni kontrolniki lahko naredijo aplikacijo neuporabno s strani končnega uporabnika, in tako lahko celoten projekt propade.

Kontrolniki so v WPF pozicionirani relativno, in niso vezani na absolutne koordinate, kar pomeni, da je postavitev neodvisna od resolucije. Postavitev bi tako lahko opisali kot rekurziven sistem, ki glede na velikost pozicionira in izriše elemente aplikacije.

2.4.1. Okna

Pri gradnji WPF-aplikacij najprej opazimo okno, ki je osnova vsake (WPF) aplikacije. V oknu so vse kontrole, ki jih potrebujemo za uporabniški vmesnik.

Vsako okno ima dva dela, in sicer:

- uporabniški del: tukaj so vse kontrole za interakcijo z uporabnikom,
- neuporabniški del: zunanja meja z gumbi za minimiziranje, ikono, sistemski meni itd. [9]

V WPF imamo na voljo tri različne tipe oken:

- Okno (ang. Window): normalno okno, kjer namestimo kontrole.
- Navigacijsko okno (ang. Navigation Window): izvira iz normalnega okna, le da ima možnost navigacije z gumbi naprej/nazaj, primerno je za razne čarovnike.
- Stran (ang. Page): podobno navigacijskemu oknu, le da to okno lahko odpremo v internetnem brskalniku kot XBAP-aplikacijo.

2.4.2. Postavitvene plošče

Dobra postavitev elementov je kritična za uporabnost aplikacije. Narediti uporaben uporabniški vmesnik, ki je tudi inovativen in privlačnega videza, je izjemno težko.

Pri doseganju tega cilja nam v WPF pomagajo različne postavitvene plošče, ki so:

- Platno
- Skladovna plošča
- Ovijalna plošča
- Odlagalna plošča
- Tabela

Običajno se le-te uporabljajo v kombinaciji ena z drugo, da dosežemo željen učinek. Prav te plošče nam omogočajo relativno pozicioniranje elementov. Statično pozicioniranje kontrol s piksli je stvar preteklosti, saj le-to zelo omejujoče. Problem se pojavi že pri različnih resolucijah, napravah, uporabniških nastavitvah (različne pisave itd.) ali spremembah vsebine okna.

Platno

Platno je najosnovnejša postavitvena plošča. V platno pozicioniramo elemente z navedbo koordinat oz točk. Slabost je torej absolutna postavitev elementov, zato se platno ne uporablja pogosto, saj še imamo na voljo naprednejše postavitvene plošče, ki so lažje in boljše za uporabo.

Skladovna plošča

Skladovna plošča (ang. Stack Panel) je zaradi svoje preprostosti in uporabnosti zelo popularna. V njej lahko pozicioniramo elemente enega za drugim, tako vertikalno kot horizontalno.

Za delo s to ploščo ni potrebnih veliko nastavitev, možno je tudi brez njih. Včasih je potrebno le nastaviti orientacijo, ki je privzeto vertikalna, kar pomeni, da pozicionira elemente vertikalno enega za drugim. Lahko pa jih tudi horizontalno.

Ovijalna plošča

Plošča, podobna skladovni, le da ta pozicionira elemente v nove vrstice oz. stolpce, glede na orientacijo.

Torej, elemente ovije in jih uvršča v vrstico, dokler element še lahko uvrsti v vrsto, če ne more, ga da v novo vrsto itd.

V njej lahko omejimo velikost posameznega elementa (ItemHeight, ItemWidth) in če je le-ta večji, ga preprosto odreže.

Odlagalna plošča

Odlagalna plošča (ang. Dock Panel) omogoča enostavno odlaganje elementov oz. kontrol po stranici plošče, tako da se razteza vertikalno ali horizontalno. Omogoča tudi, da element zapolni preostali prostor, ki ni v uporabi s strani odlagalne plošče.

Odlagalna plošča omogoča odlaganje elementov na vrhu, desni ali levi strani ter spodaj. Te nastavitve določimo z atributom DockPanel.Dock.

Postavitev elementov je odvisna od zaporedja, v katerem so zapisani v XAML-u. Na posamezno stranico pa lahko odložimo več elementov.

Uporabna je za vzpostavitev osnovnega okna aplikacije, saj ima običajno aplikacija na vrhu kakšen meni, na strani navigacijo, spodaj statusno vrstico in seveda v centru ploščo z vsebino.

Tabela

Tabela (ang. Grid) je najbolj vsestranska plošča v WPF, ki je po vsej verjetnosti tudi najbolj uporabljena s strani razvijalcev. Omogoča razporeditev elementov po vrsticah in stolpcih. Delo z WPF-tabelo je podobno kot s HTML-tabelo. Na sliki 2 smo naredili vnosno okno s pomočjo tabele. Tabele imajo več različnih nastavitev, kar se tiče velikosti celic.

Tri osnovne nastavitve:

- Absolutna: določimo višino in širino v pikslih, velikost se ne spreminja.
- Proporcionalna: to velikost določimo z zvezdico, kar pomeni, da se plošča vedno razteza v celoten prostor, ki je na voljo.
- Avtomatska: posamezna celica da elementom znotraj nje prostora toliko, kolikor le-ti rabijo; torej se avtomatsko razteza glede na notranje elemente.

Tabela omogoča tudi interaktivno spreminjanje velikosti stolpcev ali vrstic, in sicer s pomočjo GridSplitterja. GridSplitter je najbolje dati v svoj stolpec ali vrstico. Potrebno je vedeti, da kadar uporabimo glidsplitter, le-ta spremeni vse višine oz. širine v absolutne.

Posamezne tabele si lahko med seboj delijo širino oz. višino s pomočjo IsSharedSizeScope in SharedSizeGroup atributov. Prvega določimo na tabeli, v katerem imamo namen deliti velikost, slednjega pa določimo na posamezen stolpec ali vrstico. [6][9] [10][12]

2.5. Kontrolniki

WPF ima veliko zbirko že vgrajenih kontrolnikov, ki omogočajo, da kar se da hitro sestavimo željen uporabniški vmesnik. Mnoge že poznamo iz starejših in drugih tehnologij, zato jih podrobneje ne bomo obravnavali.

Kontrolnike običajno ustvarjamo v XAML, lahko jih pa tudi v programski kodi, seveda pa lahko v programski kodi dostopamo do XAML-elementov, nastaviti moramo le lastnost x:Name.

Vsaka WPF-kontrola ima štiri privzete teme oz. izglede, ki se prikažejo v odvisnosti od sistema, na katerem poganjamo aplikacijo:

- Aero (Win 7 in Vista)
- Luna (Win XP)
- Royale (Win XP Media Center)
- Classic (ostali Windows sistemi) [12]

Seveda pa lahko vsako WPF-kontrolo oz. prikaz podatka spremenimo po svojih željah s pomočjo predlog, ki jih bomo spoznali v nadaljevanju.

WPF-kontrolnike lahko razdelimo med vsebinske, zbirne, uporabniške in ostale, kot so kontrole za slike, dokumente, tekst. [12]

2.5.1. Vsebinske kontrole

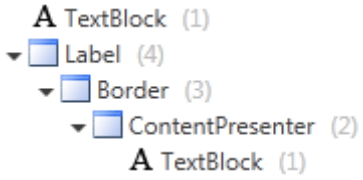
Vsebinske kontrole so kontrole, katere vsebujejo vsebino in omogočajo interakcijo s strani uporabnika.

Vzamemo za primer TextBlock in Label, na pogled enaki kontroli, obe prikazujeta tekst. Ampak TextBlock ni vsebinska uporabniška kontrola, medtem ko Label to je.

Label omogoča interakcijo z uporabnikom, in sicer z Alt in še katero drugo tipko. Tudi, če pogledamo, od kod Label in TextBlock izvirata, vidimo, da TextBlock deduje neposredno od FrameworkElement razreda, medtem ko Label izvira iz ContentControl, kar pomeni:

- naredimo lahko svojo predlogo, torej spremenimo videz,
- prikaže vsebino, ki ni le string preko Content lastnosti,
- itd. [12]

Label je med tem »težja« kontrola. Če pogledamo vizualno drevo teh dveh kontrol vidimo, da Label vsebuje TextBlock.



Slika 12: Vizualno drevo TextBlock in Label

2.5.2. Zbirne kontrole

Zbirne kontrole so tiste, ki so sposobne prikazovati zbirke podatkov. Ti zbirniki hranijo posamezen podatek iz zbirke v lastnosti Item.

V zbirne kontrole lahko damo posamezni podatek ročno v XAML, proceduralni kodi, ali pa preprosto vežemo zbirko na kontrolo s pomočjo ItemsSource lastnosti. Ta lastnost je skupna vsem kontrolam za prikaz zbirk.

Zbirka je lahko kateregakoli tipa, ki implementira IEnumerable vmesnik, torej je lahko List, ObservableCollection, ArrayList itd. Za vsako zbirno kontrolo je možno dodati sortirni kriterij za prikaz podatkov, kar pomeni, da lahko podatke sortiramo glede na lastnost, ne da bi to vplivalo na izvirno zbirko.

2.5.3. Uporabniške kontrole

Uporabniška kontrola (ang. User Control) je kontrola, ki jo uporabnik ustvari sam. Običajno je le-ta sestavljena iz standardnih WPF-kontrolnikov, vendar je zaradi večkratne uporabe narejena kot ena kontrola.

Pri ustvarjanju uporabniške kontrole so nam v veliko pomoč odvisnostne lastnosti in preusmerjeni dogodki.

Za uporabniško kontrolo ne moremo posebej ustvariti stilov in predlog.

2.6. Viri

V WPF imamo pravzaprav dve vrsti virov (ang. Resources), in sicer zbirne vire (ang. Assembly resources), ki jih poznamo že iz drugih .NET-aplikacij, ter objektne vire.

Osredotočili se bomo na slednje, imenovali pa jih bomo kar »viri«.

Viri so običajno določeni v XAML-datoteki z Resource korenskim elementom, ustvarimo jih tudi v proceduralni kodi. Lahko pa so navedeni tudi v Resource lastnosti kateregakoli elementa.

Objektni viri imajo tri glavne koristi:

- lažje vzdrževanje,
- učinkovitost,
- prilagodljivost.

Predstavljajo enostaven način ponovne uporabe objektov in vrednosti, kot so stili, predloge, elementi.

Prav tako omogočajo, da nastavimo lastnosti več kontrolam hkrati.

Tako lahko nastavimo npr. širino vseh TextBoxov hkrati s samo enim virom.

```
<Window.Resources>
    <Style TargetType="TextBlock">
        <Setter Property="Width" Value="150"/>
    </Style>
</Window.Resources>
```

Slika 13: Primer deklaracije stila za vse kontrole TextBlock

V primeru zgoraj je stil določen v kontekstu celotnega okna, kjer se viri tudi najpogosteje navajajo, lahko pa jih damo pod katerokoli drugo kontrolo ali pa v ločeno datoteko.

Sicer pa poznamo dve vrsti virov:

- Statični: za statičen vir uporabljamo StaticResource razširitev. Tem virom se vrednost določi ob nalaganju kontrole.
- Dinamični: za dinamičen vir uporabimo DynamicResource. Uporabimo ga, kadar želimo spremeniti vrednost vira med izvajanjem aplikacije.

```
<TextBlock Style="{StaticResource tbStyle}"/>
<TextBlock Style="{DynamicResource tbStyle}"/>
```

Slika 14: Uporaba vira

Posamezni viri so dostopni tudi preko proceduralne kode. Za še lažje delo pa so nam v pomoč t.i. slovarji virov (ang. Resource Dictionary). Slovar je običajna XAML-datoteka, ločena od drugih datotek, le da je korenski element tipa ResourceDictionary. S pomočjo slovarjev lahko posamezni vir uporabljamo v več pogledih oz. oknih, potrebno je le referencirati datoteko z viri.

```
<Window.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Viri.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Window.Resources>
```

Slika 15: Primer sklica na vir

V lastnosti MergedDictionaries se lahko sklicujemo na poljubno število slovarjev. Tudi v samih slovarjih se sklicujemo na druge slovarje.

Dobra praksa je, da stile, predloge in ostale vire definiramo v teh slovarjih.

2.7. Predloge in stili

Ena najboljših zmožnosti WPF-a se skriva prav v predlogah in stilih. Dvoje nam namreč omogoča, da popolnoma spremenimo izgled kontrole oz. prikaz podatkov.

Medtem ko so stili podobni CSS-funkcijam, ki jih poznamo iz HTML-sveta, so kontrolne predloge nekakšna razširitev stilov in nam omogočajo, da popolnoma spremenimo vizualno strukturo kontrole. [9][12]

2.7.1. Predloge

Podloge se v WPF pogosto uporabljajo in prav vsak razvijalec bolj naprednejše WPF-aplikacije bo uporabil kakšno.

Najpogosteje uporabljeni sta prav gotovo Kontrolna in Podatkovna predloga. To je razlog, zakaj nimajo WPF-kontrole nekaterih lastnosti za spreminjanje izgleda. Če želimo npr. spremeniti barvo gumba na ComboBoxu, moramo narediti kontrolno predlogo.

- Kontrolna predloga

Kontrolne podloge nam omogočajo, da popolnoma spremenimo vizualno drevo, torej sestavo posamezne kontrole, medtem pa ohranimo funkcionalnost.

Tako lahko npr. spremenimo izgled ComboBoxa, da ima drugačen gumb itd.

```
<Button x:Name="button" Content="Prijava" Grid.Row="2" Grid.Column="1">
    <Button.Template>
        <ControlTemplate>
            <Grid>
                <Ellipse x:Name="ii" Width="50" Height="30" Fill="LightSkyBlue"
                         Stroke="#FF10ABFF" StrokeThickness="3" />
                <TextBlock Text="Prijava" HorizontalAlignment="Center"
                           VerticalAlignment="Center"/>
            </Grid>
        </ControlTemplate>
    </Button.Template>
</Button>
```

Slika 16: Primer kontrolne predloge za gumb

- Podatkovna predloga

Podatkovne podloge nam omogočajo, da ustvarimo vizualno reprezentacijo poslovnega objekta.

Torej, s pomočjo podatkovnih podlog določimo, na kakšen način bo posamezni objekt vizualno prikazan, medtem ko s kontrolno podlogo določimo, na kakšen način bo prikazana posamezna kontrola.

Veliko WPF-kontrol ima lastnost tipa DataTemplate, na katero lahko pripnemo ustrezno podlogo. Običajno so te ContentTemplate, ItemTemplate ali pa CellTemplate, obstaja pa še veliko drugih. V primeru, da želimo spremeniti prikaz posameznega objekta v zbirni kontroli, naredimo podatkovno predlogo za ItemTemplate.

```
<DataTemplate x:Key="CubesAndDimensionsExpirationDate">
    <Border BorderThickness="1" BorderBrush="{Binding ExpirationDateColor}">
        <TextBlock Grid.Column="4" Text="{Binding ExpirationDate}"
                   Background="{Binding ExpirationDateColor}"/>
    </Border>
</DataTemplate>
```

Slika 17: Primer podatkovne predloge

HierarchicalDataTemplate

Ta vrsta podlog izvira iz podatkovnih podlog, le da so namenjene prikazu objekta in njegovih otrok. Torej, če bi želeli prikazati podatke, kot je to v TreeView, bi uporabili to podlogo. [9][12]

2.7.2. Stili

Kot že omenjeno, so WPF-stili podobni CSS-standardu v HTML-jeziku, le da imajo več zmožnosti. Omogočajo nam spremeniti izgled kontroli, ali pa tudi več kontrolam hkrati.

S stilom na sliki 13 nastavimo širino vseh TextBlock kontrol, ki se nahajajo v oknu. V primeru, da bi stil imel edinstven ključ, bi se morali sklicevati nanj v Style lastnost na kontroli s pomočjo razširitev (Slika 14).

Posameznemu TextBlocku pa lahko vseeno prepišemo določeno lastnost preprosto tako, da jo ponovno določimo v sami deklaraciji kontrole.

WPF-stili imajo tudi prožilce (ang. Triggers), ki nam omogočajo spremembo stila, kadar se spremeni kakšna druga lastnost. Znotraj njih lahko tudi definiramo predloge. [12]

Za (osnovno) definicijo stilov je potrebno poznati le nekaj lastnosti:

- Setters: nastavljajo vrednosti lastnostim z lastnostmi Property in Value.
- Triggers: prožilci, ki nam omogočajo avtomatično spreminjanje stilov.
- BasedOn: omogoča dedovanje od katerega drugega stila.
- TargetType: tip kontrole, ki jo želimo spreminjati.

```
<Style x:Key="CellStyle" TargetType="{x:Type DataGridCell}">
    <Setter Property="Background" Value="#FDFDFF" />
    <Setter Property="BorderThickness" Value="0" />
    <Style.Triggers>
        <Trigger Property="IsSelected" Value="True">
            <Setter Property="Background" Value="##560000FF" />
            <Setter Property="Foreground" Value="#E700004F" />
            <Setter Property="BorderThickness" Value="0" />
        </Trigger>
    </Style.Triggers>
</Style>
```

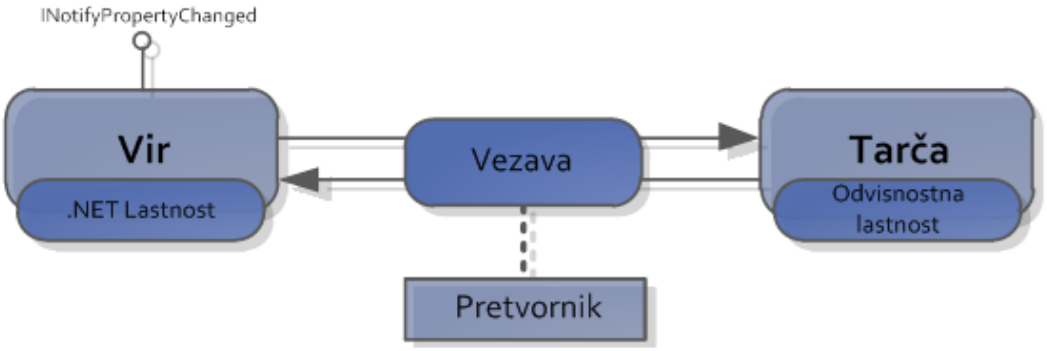
Slika 18: Primer definicije stila s prožilci

2.8. Vezanje podatkov

WPF omogoča preprost in uporaben način za avtomatsko posodabljanje podatkov med poslovnim modelom in uporabniškim vmesnikom. Vezanje podatkov torej omogoča povezavo atributa oz. lastnosti posamezne kontrole tipa DependancyProperty z. NET-objektom, sama povezava pa je lahko enosmerna ali pa dvosmerna. Kadar se spremeni podatek na poslovnem modelu, se ustrezno spremeni tudi uporabniški vmesnik. Za uspešno delovanje osvežitve vmesnika morata imeti obe strani vezave tudi ustrezne notifikacije sprememb. Na .NET-lastnostih to dosežemo z implementacijo vmesnika

INotifyPropertyChanged, medtem ko imajo odvisnostne lastnosti to obnašanje že vgrajeno. Vežemo se lahko tudi na drugi element preko imena elementa in njegove lastnosti. [10]

V primeru MVVM je običajno vir vezanja podatkov normalen .NET-objekt, medtem ko mora biti tarča odvisnostna lastnost.



Slika 19: Shema vezave podatkov [10]

Vezavo .NET-objekta z lastnostjo v XML lahko vzpostavimo preko proceduralne kode ali kar v XAML, kjer se tudi tipično vzpostavi. Za delujočo vezavo mora biti DataContext elementa tisti razred, ki to lastnost vsebuje (v tem primeru število).

```
<TextBox Text="{Binding Stevilo, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"/>
```

Slika 20: Vezanje podatkov na objekt v XAML

```
Razred st = new Razred();
Binding b = new Binding();
b.Source = st;
b.UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged;
b.Path = new PropertyPath("Stevilo");
b.Mode = BindingMode.TwoWay;
tbStevilo.SetBinding(TextBox.TextProperty, b);
```

Slika 21: Vezanje podatkov v proceduralni kodi

Sama povezava ima več lastnosti, ki definirajo obnašanje le-te, glavne med njimi so:

- Mode: način vezave (enosmeren ali dvosmeren)
- UpdateSourceTrigger: prožilec osvežitve vmesnika
- Converter: pretvornik vrednosti

V primeru, da vezava ne deluje, ne pride do izjeme, napako pa lahko vidimo v debug traces. Če vezava ne deluje, je najbolje, da program zaženemo z razhroščevalnikom in pogledamo sledi.

DataContext

Vsaka WPF-kontrola, ki deduje od FrameworkElement razreda, ima DataContext lastnost. Ta lastnost naj bi bila nastavljena na objekt, ki ga vizualizira. Vse podkontrole podedujejo ta DataContext.

DataContext specificira privzet objekt za vezavo podatkov za elemente oz. podelemente. Podelementi določenega elementa vedno podedujejo DataContext, zato je potrebo biti pazljiv pri vezavi podatkov, saj če lastnost ne obstaja na danem DataContextu, vezava ne deluje. [9][12] Pri rešitvi tega nam je v pomoč lastnost RelativeSource, s pomočjo katere se sklicujemo na DataContext druge kontrole.

```
<CheckBox IsChecked="{Binding IsSelected}"
          Margin="2"
          Command="{Binding DataContext.IsSelectedCommand,
          RelativeSource={RelativeSource FindAncestor, AncestorType=ItemsControl, AncestorLevel=1}}">
```

Slika 22: Uporaba relativnega vira

V zgornjem primeru imamo CheckBox v zbirni kontroli, kar pomeni, da je DataContext elementa Item posamezen objekt iz zbirke. Komanda se ne nahaja v tem objektu, ampak je v DataContextu zbirne kontrole (ItemsControl).

AncestorType pomeni tip nadelementa, ki ga iščemo, AncestorLevel pa je nivo, na katerem se element nahaja v primeru gnezdenja.

Vrednostni pretvorniki

Vrednostni pretvornik uporabimo v primeru, da želimo vezati dve lastnosti različnih tipov. Pretvornik dobi vhodno vrednost z vira ter vrne vrednost enakega tipa, kot je potrebovana na tarči, torej kontroli. V tem primeru ustvarimo nov razred, ki deduje od vmesnika IValueConverter ali pa IMultiValueConverter, če moramo pretvarjati več vrednosti v eno.

WPF ima tudi že narejene pretvornike, eden izmed njih je BooleanToVisibilityConverter, ki pretvori boolean tip v Visibility.

```
public class StringToIntConverter:IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        if (value != DependencyProperty.UnsetValue)
        {
            if ((int)value == 1)
            {
                return "Ena";
            }
            if ((int)value == 2)
            {
                return "Dva";
            }
        }

        return "Drugo stevilo";
    }

    public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        if (value != DependencyProperty.UnsetValue)
        {
            if ((string)value == "Ena")
            {
                return 1;
            }
            if ((string)value == "Dva")
            {
                return 2;
            }
        }

        return 0;
    }
}
```

Slika 23: Primer ustvarjenega pretvornika tipov

```
<Window.Resources>
    <converter:StringToIntConverter x:Key="StringToIntConverter"/>
</Window.Resources>

        <TextBox
            Text="{Binding Stevilo,
            UpdateSourceTrigger=PropertyChanged,
            Mode=TwoWay,
            Converter={StaticResource StringToIntConverter}}"
            />
```

Slika 24: Primer deklaracije in uporabe pretvornika v XAML

2.9. Ukazi

WPF-ukazi nam omogočijo ločevanje dogodkov uporabniškega vmesnika in obravnavo le-teh. Kar pomeni, da lahko ukaze ločimo od uporabniškega vmesnika; v primeru MVVM-vzorca jih damo v View Model, tako postanejo ukazi testabilni in lažji za vzdrževanje.

Uporabljamo jih preko lastnosti Command, ki jo vsebujejo gumbi, lahko pa seveda ustvarimo svojo, s pomočjo odvisnostnih lastnosti.

Ukazi so ena izmed temeljnih uporabljenih funkcij v MVVM-modelu, sicer lahko WPF-aplikacijo naredimo tudi popolnoma brez njih.

WPF že vsebuje nekatere vnaprej narejene osnovne ukaze v obliki statičnih razredov, ki so razdeljeni v pet kategorij:

- - ApplicationCommands
- - ComponentCommands
- - MediaCommands
- - NavigationCommands
- - EditingCommands

Svoje implementiramo z uporabo vmesnika ICommand. Posamezen ukaz ima lahko tudi parameter, ki ga priredimo s pomočjo lastnosti CommandParameter. Seveda pa obstaja več načinov, kako ustvariti svoj ukaz, eden izmed osnovnejših je prikazan v nadaljevanju.

```
public class KlikCommand:ICommand
{
    private MainWindowViewModel _viewModel;
    public KlikCommand(MainWindowViewModel viewModel)
    {
        this._viewModel = viewModel;
    }

    public bool CanExecute(object parameter)
    {
        return
            !string.IsNullOrWhiteSpace(_viewModel.UporabniškoIme)
            && !string.IsNullOrWhiteSpace(parameter as string) ? true : false;
    }

    public event EventHandler CanExecuteChanged;
    public void Execute(object parameter)
    {
        _viewModel.Prijava(parameter as string);
    }
}
```

Slika 25: Definicija ukaza

Tole je torej osnovna definicija posameznega ukaza. V večjih aplikacijah seveda ta implementacija ni tako priročna, saj je v primeru velikega števila ukazov tudi veliko kode; tako postane samo definiranje zamudno. Rešitev se skriva v implementaciji osnovnega razreda za ukaze, po navadi imenovanega DelegateCommand ali pa RelayCommand, o čemer bomo razmišljali v naslednjih odstavkih.

2.10. Orodja

Za izdelavo večjih WPF-aplikacij običajno potrebujemo več orodij, sploh če imamo razvijalce ločene na tiste, ki pišejo logiko in tiste, ki izdelujejo grafično podobo aplikacije.

V glavnem so v uporabi naslednje aplikacije:

a. Microsoft Visual Studio

Visual studio je, tako kot tudi pri izdelavi aplikacij drugega tipa, osnovna programska oprema za izdelavo WPF-aplikacij. V njej lahko pišemo proceduralno kodo, kot tudi izdelujemo grafično podobo aplikacije.

Namenjena je sicer bolj pisanju proceduralne kode, saj je po drugi strani delo na grafiki nekoliko lažje z aplikacijo Expression Blend.

b. Microsoft Expression Blend

Expression Blend je bolj namenjen delu z grafičnim delom WPF-aplikacij.

Omogoča nam lažje upravljanje s predlogami, z vezanjem podatkov in nasploh s postavljanjem kontrol. Veliko lažje je tudi ustvarjanje kontrolnih predlog.

Mnogokrat se uporablja kombinacija Blenda in Visual Studia, spremembe v enem so prepoznane v drugem.

c. WPF-Inspector

WPF-Inspector je odprto kodno orodje, ki se pripne na zagnano WPF-aplikacijo. Uporabna je predvsem za odpravljanje težav glede postavitve kontrol, z vezanjem podatkov in stilov. Pogledamo lahko tudi vizualno in logično drevo aplikacije oz. kontrol, za lažjo izdelavo kontrolnih predlog. [12]

3. MODEL-POGLED-MODEL POGLEDA (MODEL-VIEW-VIEW MODEL)

Odkar se že razvija programska oprema z uporabniškim vmesnikom, obstajajo tudi načrtovalski vzorci, ki to delo olajšujejo.

Med bolj popularnimi je npr. MVP (ang. Model-View-Presenter), ki je pravzaprav variacija vzorca MVC (Model-View-Controller), ki je izjemno popularen za gradnjo ASP.NET-aplikacij in obstaja že desetletja.

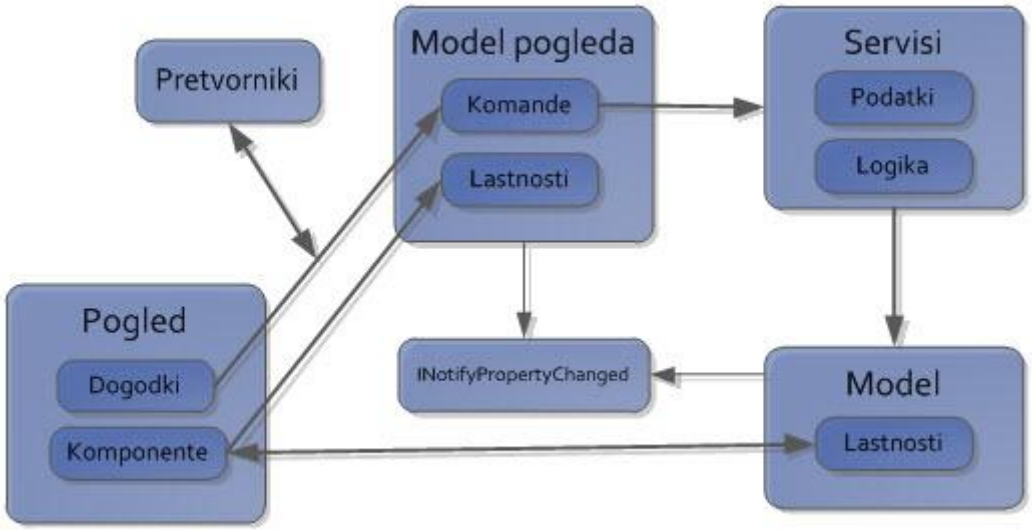
Potem je tudi PM (Presentation Model), ki je podoben MVP in loči pogled od obnašanja in stanja. Zanimiv del tega vzorca je, da ustvarimo abstrakcijo pogleda, imenovan predstavitev modela, ki sproti posodablja pogled, tako da sta ta dva dela sinhronizirana.

MVVM je identičen temu vzorcu glede funkcij in abstrakcije pogleda ter je pravzaprav specializacija tega bolj splošnega vzorca za WPF in Silverlight, tudi izjemno popularnega s strani razvijalcev. Torej MVVM v bistvu sploh ni kaj veliko novega, saj je ločitev pogleda od logike že dolgo poznan koncept pri razvoju aplikacij z uporabniškim vmesnikom. [14]

Seveda imamo tudi s tem vzorcem, kot z vsakim drugim, več možnih variacij in deljenih mnenj glede implementacije. Lahko si tudi pomagamo z raznimi orodji oz. knjižnicami, ki nam olajšajo implementacijo.

MVVM vsebuje tri ključne komponente, ki jih bomo predstavili v nadaljevanju:

- Model
- Pogled
- Model pogleda



Slika 26: Struktura MVVM-vzorca

Ključne funkcije WPF, ki smo jih tudi predelali v preteklih poglavjih in nam omogočajo implementacijo MVVM-arhitekture, so:

- Vezava podatkov (INotifyPropertyChanged)
  - Lastnosti imajo prožilce za notifikacijo sprememb, da se vmesnik ustrezno osvežuje.
- Ukazi (ICommand)
  - Ukazi so vezani na gumbe.
- Odvisnostne lastnosti
- Vrednostni pretvorniki (IValueConverter, IMultiValueConverter)

Najpomembnejša med njimi je prav gotovo vezava podatkov, ki nam omogoča interakcijo med pogledom in modelom pogleda.

Glavne koristi gradnje aplikacije po MVVM-arhitekturi so:

- Ločitev skrbi (ang. Seperation of concerns).

- Omogoči testiranje.
- Injiciranje odvisnosti.
- Spodbuja delovni tok Razvijalec-Oblikovalec.
- Lažje spreminjanje na uporabniškem vmesniku ali logike pozneje v razvoju.
- Podatki v modelu so lahko drugačni kot tisti na uporabniškem vmesniku, saj lahko s pogledom modela npr. agregiramo določene podatke.
- Lažje osveževanje uporabniškega vmesnika.
- Lažja razširljivost aplikacije.

Za lažjo implementacijo MVVM-vzorca je najbolje implementirati osnovna razreda za ukaze, ki implementira vmesnik ICommand, in pogled modela, ki implementira vmesnik INotifyPropertyChanged, kot je to storjeno v nadaljevanju, saj je v nasprotnem primeru veliko več kode.

3.1. Model

Model predstavlja dejanske podatke in informacije, ki jih obravnavamo. To je lahko npr. oseba, ki vsebuje ime, priimek in naslov. [5]

Model je torej običajen .NET-razred, ki pogledu modela zagotavlja podatke ali pa predstavlja le entitete z lastnostmi.

Potrebno si je zapomniti, da le drži informacije, ne pa obnašanja, logike za manipulacijo s podatki, tudi podatkov ne jemlje iz baze. Posamezen podatek iz baze pa je po vsej verjetnosti model. Poslovna logika je običajno ločena od modela v svojih razredih. Seveda pa se poslovni logiki, sploh v večjih projektih, ne da popolnoma izogniti. [5]

3.2. Pogled

Pogled je pravzaprav XAML-datoteka, v kateri je definirana struktura, izgled in interakcija z uporabnikom. V WPF je to običajno lahko okno, uporabniška kontrola ali pa stran. V kodi pogleda je zelo malo ali nič proceduralne kode; le-ta je večina v modelu pogleda.

Glede vsebnosti kode v razredu pogleda je bilo že veliko debate, izogniti se ji je seveda vedno možno, ampak je lahko takšno početje časovno zelo potratno. V primeru kode, naj bo ta omejena le na interakcijo s pogledom, poslovni logiki, se je potrebno izogibati, zlasti, če želimo testabilno in za vzdrževanje lahko aplikacijo.

V pogledu se uporabljajo vezava podatkov, vrednostni pretvorniki in ukazi. Nastaviti je potrebno lastnost DataContext, in sicer na model pogleda.

```
<Window x:Class="DNPrimer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:c="clr-namespace:LogicalVisualTree"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <c:MainWindowViewModel x:Key="MainWindowViewModelDataSource" d:IsDataSource="True"/>
    </Window.Resources>

    <Grid DataContext="{Binding Source={StaticResource MainWindowViewModelDataSource}}">
        <Border CornerRadius="6" BorderBrush="AliceBlue" BorderThickness="3"/>
        <Grid HorizontalAlignment="Center" VerticalAlignment="Center" Height="120" Width="250">
            <Grid.Resources...>
            <Grid.RowDefinitions...>
            <Grid.ColumnDefinitions...>
            <TextBlock Text="Uporabniško ime:" Grid.Row="0" Grid.Column="0" />
            <TextBlock Text="Geslo:" Grid.Row="1" Grid.Column="0" />
            <TextBox  Grid.Row="0" Grid.Column="1" Text="{Binding UporabniškoIme, Mode=TwoWay}" />
            <PasswordBox x:Name="passwordBox"  Grid.Row="1" Grid.Column="1" />
            <Button x:Name="button" Content="Prijava" Grid.Row="2" Grid.Column="1" Command="{Binding Klik}"/>
        </Grid>
    </Grid>
</Window>
```

Slika 27: Primer enostavnega pogleda z vezavo podatkov in ukazi

3.3. Model pogleda

Model pogleda je ključna komponenta MVVM in je vmesna abstrakcija med pogledom in modelom.

Prav s pomočjo te komponente ločimo pogled od poslovne logike, saj se ukazi, metode in lastnosti nahajajo tukaj, in ne v kodi pogleda. Model pogleda zagotavlja pogledu podatke in v primeru, da naš pogled potrebuje npr. validacijo podatkov, storimo to prav tukaj.

Običajno model pogleda vsebuje ukaze, lastnosti in implementira osnovni razred modela pogleda, kot je to prikazano na sliki spodaj.

```
public class MainWindowViewModel : ViewModelBase<MainWindowViewModel>
{
    private string _uporabniškoIme = string.Empty;
    public string UporabniškoIme
    {
        get { return _uporabniškoIme; }
        set { _uporabniškoIme = value; OnPropertyChanged("UporabniškoIme"); }
    }

    DelegateCommand<string> _prijavaCommand = null;
    public DelegateCommand<string> PrijavaCommand
    {
        get
        {
            if (_prijavaCommand == null)
                _prijavaCommand = new DelegateCommand<string>(Prijava);
            return _prijavaCommand;
        }
    }

    void Prijava(string geslo)
    {
        var prijavaService = new PrijavaService();
        prijavaService.Prijava(UporabniškoIme, geslo);
    }
}
```

Slika 28: Primer modela pogleda

Model pogleda komunicira s pogledom s pomočjo vezave podatkov, lastnosti, dogodkov.

Delo z zbirkami

Za delo z zbirkami se v primeru vezave podatkov priporoča uporaba razreda ObservableCollection, ki ima že vgrajen sistem za obvestilo sprememb. Obvestilo se sproži v primeru dodajanja, odstranjevanja ali osvežitve zbirke.

Tudi vezava poteka veliko hitreje, kot recimo z razredom List, sploh z velikim številom podatkov.

4. OGRODJE ZA UPRAVLJANJE RAZŠIRITEV

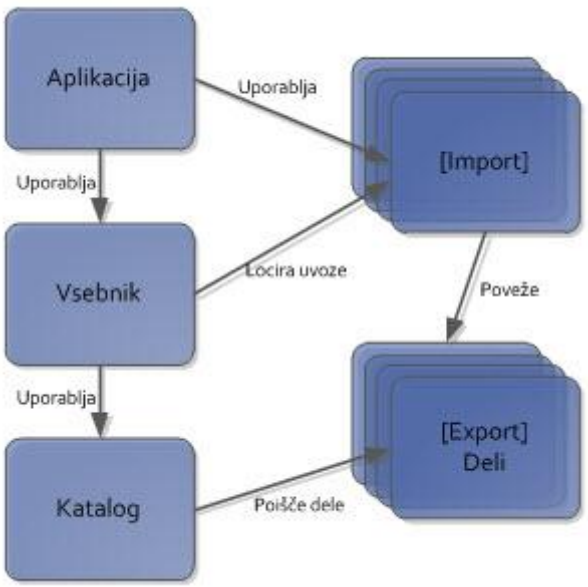
Ogrodje za upravljanje razširitev (ang. Managed Extensibility Framework) je komponenta .NET 4, ki nam omogoča gradnjo lahkih in razširljivih aplikacij. Razvijalcem aplikacij dovoljuje, da odkrijejo in uporabljajo razširitve brez predhodne konfiguracije. [7]

Z MEF si lahko aplikacijo predstavljamo kot sestavljenko, grajeno iz delov, ki jih je potrebno sestaviti skupaj, da tvorijo celoto. Lep primer so tudi spletni brskalniki, ki imajo vrsto vtičnikov (ang. Plugins), le-ti omogočajo razširitev funkcionalnosti brskalnika.

V osnovi sledi principu odprtega-zaprtega (ang. Open Close Principle), ki veleva, da morajo biti programske entitete (razredi, moduli itd.) odprte za razširitve, ampak zaprte za modifikacije.

Paradigma MEF je grajena na ideji potreb in delov, ki so lahko odkriti, da te potrebe zadovoljijo. [3] Zato ima naslednje ključne besede:

- Uvoz (ang. Import): definira potrebo v obliki pogodbe, ki mora biti izpolnjena za uspešno sestavo.
- Izvoz (ang. Export): izpostavi dele za odkrivanje.
- Katalog (ang. Catalog): vsebuje dele (izvoze).
- Sestava (ang. Compose): sestavo izvrši MEF, in tako sestavi dele, ki tvorijo celoto.



Slika 29: Arhitektura MEF

Kot je razvidno s slike 29 vsebnik uporablja katalog za iskanje delov. Katalogi služijo kot navodila za iskanje delov oz. izvozov.

MEF ima možnost uporabe več možnih vrst katalogov:

- Katalog tipov (ang. Type Catalog) – v procesu iskanja izvozov so locirani le določeni tipi.
- Imenikov katalog (ang. Directory Catalog) – dele išče v določenem imeniku nad aplikacijsko domeno. Torej v dll datotekah v določenem imeniku.
- Zbirni katalog (ang. Assembly Catalog) – išče dele v določenem zbirniku.
- Agregiran katalog (ang. Aggregate Catalog) – je pravzaprav katalog katalogov, saj lahko vanj damo vse kataloge, ki jih lahko med izvajanjem tudi dodajamo/odstranjujemo. [8]

```
static DirectoryCatalog directoryCatalog = null;
static TypeCatalog typeCatalog = null;
static AssemblyCatalog assemblyCatalog = null;
static CompositionContainer container = null;
static AggregateCatalog aggregateCatalog = null;

public static void Compose(object o, string path)
{
    try
    {
        assemblyCatalog = new AssemblyCatalog(Assembly.GetEntryAssembly());
        directoryCatalog = new DirectoryCatalog(path);
        typeCatalog = new TypeCatalog(typeof(IOseba));
        aggregateCatalog = new AggregateCatalog(directoryCatalog, assemblyCatalog,typeCatalog);
        container = new CompositionContainer(aggregateCatalog);
        container.ComposeParts(o);
    }
    catch (Exception exc)
    {
        Debug.Write("Composition error! Message: "+exc.Message);
    }
}
```

Slika 30: MEF – Sestava delov z uporabo vseh katalogov

Na sliki 30 vidimo primer metode za sestavo delov, kjer so v uporabi vse vrste katalogov.

Metoda za sestavo delov se običajno kliče kar v kakšnem vhodnem razredu v aplikaciji, ki vsebuje kakšen uvoz. MEF se nato rekurzivno sprehodi skozi uvoze v aplikaciji in jih poveže z izvozom.

Razred lahko uvozimo oz. izvozimo na tri načine, in sicer običajno le z atributom [Export], po tipu ali po imenu pogodbe.

```
public App()
{
    MEF.Compose(this);
}

[Import(typeof(MainWindow))]
public MainWindow MainWindowView { get; set; }

private void Application_Startup(object sender, StartupEventArgs e)
{
    MainWindow = MainWindowView;
    MainWindow.Show();
}
```

Slika 31: Primer vhoda z uvozom in s sestavo

```
[Export]
[Export(typeof(MainWindow))]
[Export("MainWindowView")]
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private IViewModel _viewModel;
    [Import("MainWindowViewModel",typeof(IViewModel))]
    public IViewModel ViewModel
    {
        get { return _viewModel; }
        set
        {
            _viewModel = value;
            DataContext = value;
        }
    }
}
```

Slika 32: Izvoz pogleda in nastavitev modela pogleda

Zgornja slika prikazuje vse vrste izvoza, uporabimo le enega, v nasprotnem primeru MEF sproži izjemo.

Privzeto se deli uvozijo po vzorcu Edinca (ang. Singleton), kar pomeni, da imamo le eno instanco posameznega razreda, kjerkoli ga uvažamo. Seveda lahko to obnašanje spremenimo z atributom PartCreationPolicy, ki ga nastavimo na:

- CreationPolicy.Shared: privzeta nastavitev,
- CreationPolicy.NonShared: naredi nove instance vsakič, ko uvozimo del.

Zanimivost MEF je tudi, da lahko aplikacijo posodobimo oz. razširimo kar med izvajanjem le-te. S kombinacijo uporabe imeniškega kataloga in implementacije mehanizma za zaznavanje sprememb v imeniku lahko dele ponovno sestavimo, kadar je v imenik dodan nov .dll, in tako razširimo aplikacijo. Poskrbeti moramo le, da je atribut za uvoz dodatno opremljen z atributoma AllowDefault in AllowRecomposition.

S tem povemo, da je za uvoz začasno lahko ničeln (ang. Null) ter da je lahko znova sestavljen. V primeru, da MEF ne najde izvoza, ta javi napako.

Posamezne dele lahko tudi uvozimo preko konstruktorja, in sicer z atributom ImportingConstructor.

```
[Export(typeof(MainWindow))]
public partial class MainWindow : Window
{
    [ImportingConstructor]
    public MainWindow(IViewModel viewModel)
    {
        InitializeComponent();
        DataContext = viewModel;
    }
}
```

Slika 33: Injiciranje odvisnosti preko konstruktorja

MEF je poskrbel tudi za uvoz več instanc istega tipa z atributom ImportMany, kar pomeni, da lahko v zbirko uvozimo vse dele enakega tipa.

```
[ImportMany]
IEnumerable<ILog> Loggers { get; set; }
```

Slika 34: Množičen uvoz

V primeru na sliki 34 se v zbirko uvozijo vsi deli, ki implementirajo vmesnik ILog.

Do sedaj obravnavani uvozi se vsi uvozijo ob inicializaciji primerka razreda. Kaj pa, če ne želimo dela uvažati, ker nismo prepričani, če bo sploh uporabljen? Za takšne primere ima MEF rešitev v atributu Lazy, ki nam omogoča inicializacijo lastnosti, kadar le-to potrebujemo. Del ni uvožen, dokler ni na lastnosti klicana lastnost Value.

```
[Import]
Lazy<IMessageService> ErrorMessageService { get; set; }

void Prijava(string geslo)
{
    try
    {
        var prijavaService = new PrijavaService();
        prijavaService.Prijava(UporabniškoIme, geslo);
    }
    catch (Exception ex)
    {
        var errorMessage = ErrorMessageService.Value;
        errorMessage.LogError(ex.Message, DateTime.Now.ToString("0:MM/dd/yy H:mm:ss"));
    }
}
```

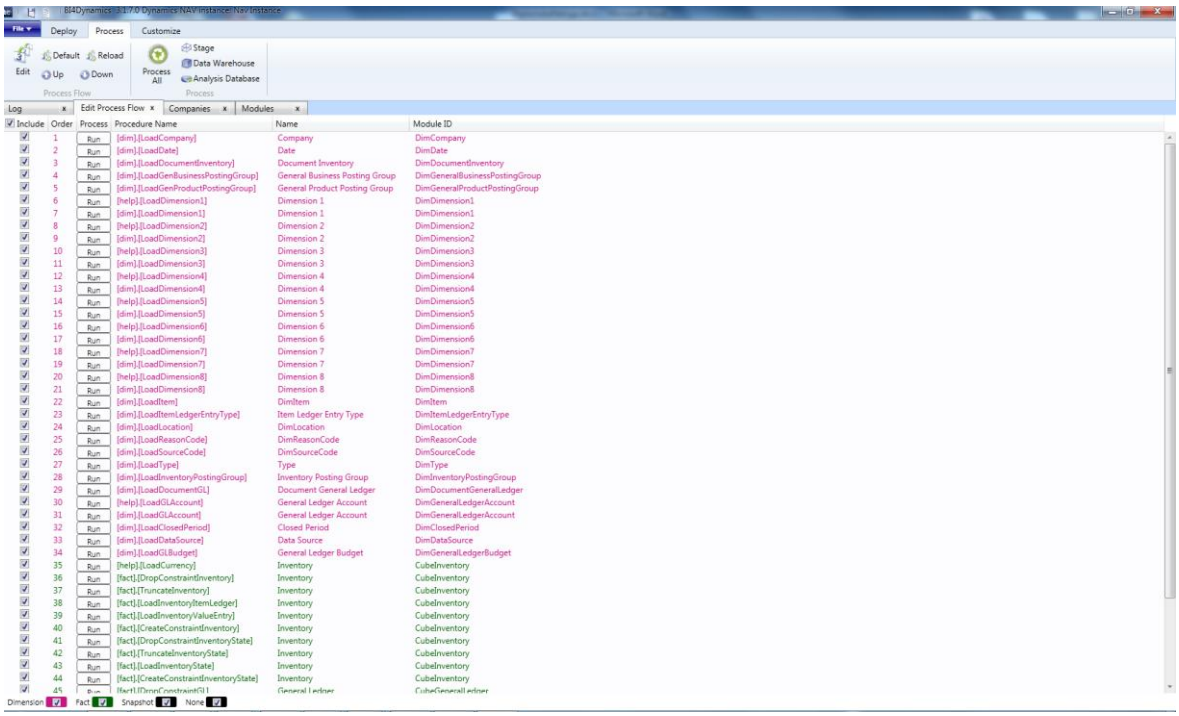
Slika 35: Len uvoz del

5. PRIMER PRAKTIČNE UPORABE

Za predstavitev zgoraj opisanih tehnologij bomo uporabili del aplikacije BI4Dynamics. BI4Dynamics je analitična rešitev poslovne inteligence za transakcijski sistem Dynamics Navision in Dynamics Axapta.

Aplikacija je v osnovi grajena v tehnologiji WPF z uporabo MVVM in ogrodja MEF. Ima dve osnovni kontroli, in sicer Ribbon ter TabControl, v slednjo se uvažajo posamezni pogledi.

Zaradi obširnosti aplikacije je v nadaljevanju predstavljen le en pogled z modelom pogleda.



Slika 36: Aplikacija BI4Dynamics

5.1. Pogled

Kot rečeno, ima aplikacija kontrolnik TabControl, kamor se uvažajo posamezni pogledi, kot vidimo na sliki 36.

Narediti je bilo potrebno pogled za prikaz in urejanje posameznih procedur v kontrolniku DataGrid. Posamezne celice so definirane s pomočjo podatkovnih predlog kar znotraj samega DataGrida. Lahko bi jih definirali posebej v virih, ampak le-to ni potrebno, saj predloge niso posebaj dolge.

DataContext je določen s pomočjo DataTemplate. DataTemplate asociira podatkoven tip ProcessFlowViewModel s samim pogledom. Sama TabControla je vezana na zbirko teh modelov pogleda in kadar je le-ta dodan v zbirko, se odpre tudi ustrezen pogled.

```
<DataTemplate DataType="{x:Type viewModels:ProcessFlowViewModel}">
    <commonView:ProcessFlowView />
</DataTemplate>
```

Slika 37: Povezava pogleda s pogledom modela

```
<Grid Background="White">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="25"/>
    </Grid.RowDefinitions>
    <Grid x:Name="DataGridGrid" MouseLeftButtonUp="DataGridGrid_MouseLeftButtonUp" Grid.Row="0" Background="White"
          DataContext="{Binding Instance.ProcessExecutionPlan, UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}" MouseMove="DataGridGrid_MouseMove" HorizontalAlignment="Left">
    <DataGrid x:Name="itemsDataGrid"
              ItemsSource="{Binding Steps}"
              SelectedItem="{Binding DataContext.SelectedStep, RelativeSource={RelativeSource FindAncestor, AncestorType=UserControl, AncestorLevel=1}, UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}"
              PreviewMouseLeftButtonDown="DataGrid_PreviewMouseLeftButtonDown"
              Style="{StaticResource DataGridStyle}"
              >
            <DataGrid.Columns>
                <DataGridTemplateColumn Width="auto">
                    <DataGridTemplateColumn.Header>
                        <CheckBox IsChecked="{Binding DataContext.IncludeAll, Mode=OneWay,
                                RelativeSource={RelativeSource FindAncestor, AncestorType=UserControl, AncestorLevel=1}}" Content="Include"
                                 Command="{Binding DataContext.IncludeSelectAllChangeCommand,
                                RelativeSource={RelativeSource FindAncestor, AncestorType=UserControl, AncestorLevel=1}}"  />
                    </DataGridTemplateColumn.Header>
                    <DataGridTemplateColumn.CellTemplate>
                        <DataTemplate>
                            <CheckBox IsChecked="{Binding Path=IsIncluded, UpdateSourceTrigger=PropertyChanged, Mode=TwoWay}" Command="{Binding DataContext.StepStateChangeCommand,
                                RelativeSource={RelativeSource FindAncestor, AncestorType=UserControl, AncestorLevel=1}}" HorizontalAlignment="Center" VerticalContentAlignment="Center"/>
                        </DataTemplate>
                    </DataGridTemplateColumn.CellTemplate>
                </DataGridTemplateColumn>

                <DataGridTemplateColumn Header="Order" Width="auto">
                    <DataGridTemplateColumn.CellTemplate>
                        <DataTemplate>
                            <TextBlock Text="{Binding SequenceNumber}" Style="{StaticResource TextBlockStyle}"
                                       Foreground="{Binding ModuleType, Converter={StaticResource ModuleTypeToColor}, Mode=OneTime}"/>
                        </DataTemplate>
                    </DataGridTemplateColumn.CellTemplate>
                </DataGridTemplateColumn>

                <DataGridTemplateColumn Header="Process" Width="auto">
```

Slika 38: XAML pogleda

Uporabili smo tudi vir (ang. ResourceDictionary), kamor smo premaknil stile. To nam naredi sam XAML; veliko preglednejši in lažji za vzdrževanje ter modificiranje. Povsod v XAML se lahko sklicujemo na poljubno število virov.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:src="clr-namespace:BI4Dynamics.Common.Converters"
                    >

    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="/BI4Dynamics.Common;component/Resources/AppColors.xaml" />
    </ResourceDictionary.MergedDictionaries>

    <!--Converters-->
    <src:ModuleTypeToColorConverter x:Key="ModuleTypeToColor" />

    <!--Styles-->
    <Style x:Key="TextBlockStyle" TargetType="TextBlock">
        <Setter Property="Foreground"
                Value="{Binding ModuleType, Converter={StaticResource ModuleTypeToColor}, Mode=OneTime}" />
        <Setter Property="Margin" Value="8,0,20,0"/>
        <Setter Property="TextWrapping" Value="NoWrap"/>
        <Setter Property="TextTrimming" Value="CharacterEllipsis"/>
    </Style>

    <Style x:Key="FotterTbStyle" TargetType="TextBlock">
        <Setter Property="Margin" Value="5,0,3,0"/>
        <Setter Property="VerticalAlignment" Value="Center"/>
        <Setter Property="FontSize" Value="11"/>
    </Style>

    <Style x:Key="GridViewCellStyle" TargetType="{x:Type DataGridCell}">
        <Setter Property="Background" Value="{StaticResource WhiteBlueTan}" />
        <Setter Property="BorderThickness" Value="0" />
        <Style.Triggers>
            <Trigger Property="IsSelected" Value="True">
                <Setter Property="Background" Value="#A0CFEC" />
                <Setter Property="BorderThickness" Value="0" />
            </Trigger>
        </Style.Triggers>
    </Style>
```

Slika 39: Vir

```
<UserControl.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="/BI4Dynamics.Common;component/Resources/ProcessFlowResources.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</UserControl.Resources>
```

Slika 40: Sklic na vir v pogled

S slike 36 je tudi razvidno, da imajo nekatere vrstice druge barve, kar sicer pomeni drugo vrsto procedure. Barvo teksta smo določili s pomočjo vrednostnega pretvornika, kjer se preveri tip procedure in tudi vrne ustrezna barva za tekst.

```
public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
{
    //return System.Windows.Media.Brushes.Black;
    if (!(value is ModuleType))
    {
        return System.Windows.Media.Brushes.Black;
    }
    switch ((ModuleType)value)
    {
        case ModuleType.Snapshot: return System.Windows.Media.Brushes.Black;
        case ModuleType.Dimension:
            return System.Windows.Media.Brushes.MediumVioletRed;
        case ModuleType.Fact:
            return System.Windows.Media.Brushes.DarkGreen;
        case ModuleType.None: return System.Windows.Media.Brushes.Black;
        default: return System.Windows.Media.Brushes.Black;
    }
}

public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
{
    return DependencyProperty.UnsetValue;
}
```

Slika 41: Vrednostni pretvornik

5.2. Model pogleda

Seveda zraven pogleda potrebujemo tudi model pogleda, ki našemu pogledu zagotavlja podatke. Za vezavo na DataGrid uporabimo zbirko ObservableCollection, ki sama poskrbi za osveževanje pogleda ob dodajanju in odstranjevanju iz nje.

Sam vir podatkov za DataGrid se ne nahaja v pogledu modela, ampak imamo tam le lastnost Instance, ki med drugim vsebuje to zbirko, zato določimo DataContext tudi na kontrolniku Grid. Na CheckBox kontrolnikih znotraj DataGrida je razvidno, da uporabljamo RelativeSource, ki se sklicuje na UserControl, katerega DataContext je sam model pogleda, kjer je tudi definiran posamezen ukaz. V primeru, da ukaza ne kličemo z RelativeSource, WPF le-tega išče kar v DataContextu posameznega podatka znotraj zbirke, seveda ga tam ne bi našel.

```
//[Export("ProcessFlowViewModel")]
[ViewAttribute(typeof(ProcessFlowViewModel))]
public class ProcessFlowViewModel : WorkspaceViewModel, ITabView
{
    public ProcessFlowViewModel()
    {
        TabCaption = "Edit Process Flow";
        //task = null;
    }

    properties

    [Import("AppInstance")]
    AppInstance App { get; set; }

    [Import("LoggerService")]
    public LoggerService Logger { get; set; }

    #region commands
    DelegateCommand _stepStateChangeCommand = null;
    public DelegateCommand StepStateChangeCommand
    {
        get
        {
            if (_stepStateChangeCommand == null)
                _stepStateChangeCommand = new DelegateCommand(OnStepStateChangeCommand);
            return _stepStateChangeCommand;
        }
    }
    void OnStepStateChangeCommand()
    {
        RefreshComboboxes();
        Instance.IsDirty = true;
        OnPropertyChanged("IncludeAll");
        App.ViewModelLocator.RefreshAllViews("Instance");
    }
```

Slika 42: Model pogleda aplikacije

Kot vidimo na sliki 42, ima model pogleda ukaze in uvaža dnevnik (Logger) ter podatke, ki jih potrebujemo globalno v celotni aplikaciji (App). Ti dve lastnosti uvažajo prav vsi modeli pogleda v aplikaciji.

Sam pogled modela se izvaža z lastno narejenim atributom ViewAttribute, saj smo pri izvozu uporabili tudi meta podatke, s pomočjo katerih identificiramo sam pogled.

Ukazi so implementirani s pomočjo osnovnega razreda za ukaze (DelegateCommand). V XAML se vežemo na ukaz (StepStateChangeCommand) ter s tem ob kliku v ComboBox prožimo metodo OnStepStateChangeCommand.

Prednost uvoza z MEF je v tem primeru to, da imamo povsod uvoženo enako instanco (singleton), in tako enake podatke. Težava se zna pojaviti le z osveževanjem uporabniškega vmesnika, kadar se posamezna lastnost spremeni. V ta namen smo implementirali razred, imenovan ViewModelLocator, kjer množično in »leno« uvozimo vse modele pogleda (tipa ITabView). Tam so med drugim implementirane metode za osveževanje želenega pogleda oz. pogledov. Na sliki 42 vidimo primer uporabe le-tega, kjer na vseh pogledih osvežimo vezave na lastnosti imenovani »Instanca«. V primeru, da tega ne bi prožili in bi spremenili podatke v tej lastnosti, se drugi pogledi ne bi osvežili z novimi podatki.

```
[Export(typeof(IViewModelLocator))]
public class ViewModelLocator:IViewModelLocator
{
    [ImportMany]
    private IEnumerable<Lazy<IView, IViewAttribute>> _views { get; set; }
```

Slika 43: Uvoz modelov pogleda

```
[Export("LoggerService")]
public class LoggerService : ViewModelBase<LoggerService>
{
```

Slika 44: Izvoz dnevnika

Skozi ta primer je torej prikazana uporaba vseh pomembnejših lastnosti, ki smo jih omenili v prejšnjih poglavjih, in sicer od stilov pa do uporabe ukazov.

6. SKLEP

V diplomski nalogi je predstavljena tehnologija za razvoj uporabniško bogatih namiznih aplikacij, imenovana WPF.

Sama grafična podoba aplikacije pa seveda, vsaj z razvojnega vidika, ni najpomembnejša. Aplikacije je običajno potrebno vzdrževati, zahtevajo pa tudi stalno nadgrajevanje, tako uporabniškega vmesnika kot funkcij aplikacije.

Da bi to delo olajšali, smo v drugem poglavju obravnavali arhitekturen vzorec MVVM. Z delom po tem vzorcu poskrbimo, da je logika ločena od vmesnika, in tako posledično postane aplikacija testabila. Obenem pa tudi poskrbi za lažje osveževanje uporabniškega vmesnika s pomočjo vezave podatkov. Poenostavi se tudi sam razvoj, saj se lahko delo loči na razvijalce poslovne logike in oblikovalce grafičnega vmesnika, tako se delo opravi hitreje ter posledično tudi ceneje.

Za lažjo razširitev in splošen razvoj poskrbimo s pomočjo ogrodja MEF, spoznanem v zadnjem poglavju, s čimer lahko izvažamo in uvažamo nove funkcionalnosti v naši aplikaciji.

S temi tremi tehnologijami lahko torej ustvarimo uporabne, robustne in razširljive aplikacije, ki so tudi lahke za vzdrževanje.

Vse te prednosti pa ostanejo le teoretične, če aplikacija ni dobro načrtovana in pravilno implementirana.

7. VIRI, LITERATURA

[1] Abrams, B. (29. 9. 2008). Simple Introduction to Extensible Applications with the Managed Extensions Framework. Pridobljeno 10. 6. 2011 s http://blogs.msdn.com/b/brada/archive/2008/09/29/simple-introduction-to-composite-applications-with-the-managed-extensions-framework.aspx

[2] Brown, P. (2010). Silverlight 4 in action. Kraj: Manning.

[3] Eshet, B. (2011). MEF for begginer. Pridobljeno 10. 6. 2011 s http://blogs.microsoft.co.il/blogs/bnaya/archive/2010/01/09/mef-for-beginner-toc.aspx

[4] Hall, G. (2010). Pro WPF & Silverlight MVVM. Kraj: Apress.

[5] Likness, J. (14. 4. 2010). Model-View-ViewModel (MVVM) Explained. Pridobljeno 8. 8. 2011 s http://csharperimage.jeremylikness.com/2010/04/model-view-viewmodel-mvvm-explained.html

[6] MacDonald, M. (2008). Pro WPF in C# 2008. Kraj: Apress.

[7] Managed Extensibility Framework. Pridobljeno 10. 8. 2011 s http://en.wikipedia.org/wiki/Managed_Extensibility_Framework

[8] MEF Community site. Codeplex. Pridobljeno 10. 6. 2011 s http://mef.codeplex.com/

[9] Microsoft. (2010). Windows Presentation Foundation, Microsoft. Pridobljeno 13. 6. 2011 s http://msdn.microsoft.com/en-us/library/ms754130.aspx

[10] Mosers, C. (2011). WPF Tutorial. Pridobljeno 9. 6. 2011 s http://www.wpftutorial.net

[11] Nagel, C. in Evjen, B. (2010). C# 4 and .NET 4. Kraj: Wrox.

[12] Nathan, A. (2010). WPF 4 Unleashed. Kraj: Sams.

[13] Nipun, T. (9. 6. 2011). WPF Logical and Visual Trees. Pridobljeno 5. 8. 2011 s http://www.c-sharpcorner.com/Blogs/5394/wpf-logical-and-visual-trees.aspx

[14] Smith, J. (Februar 2009). WPF Apps With The Model-View-ViewModel Design Pattern. MSDN Magazine. Pridobljeno 9. 6. 2011 s http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

8. PRILOGE

8.1. Naslov študenta

Ime in priimek: Matej Spindler

Naslov: Kraigherjeva 20

Pošta: 2250 Ptuj

Tel. študenta: 051/413-210

E-mail študenta: matej.spindler@gmail.com

8.2. Kratek življenjepis

Rojen: 6. 4. 1987 na Ptuju

Šolanje:

- Osnovna šola Ljudski vrt
- Srednja poklicna in tehniška elektro šola Ptuj – Računalniški tehnik
- Fakulteta za elektrotehniko, računalništvo in informatiko – Informatika in tehnologije komuniciranja; smer Razvoj informacijskih sistemov

Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a MATEJ SPINDLER,

z vpisno številko E1008793,

sem avtor/-ica diplomskega dela z naslovom:

RAZVOJ NAMIZNIH APLIKACIJ V WPF Z ARHITEKTURNIM VZORCEM MVVM IN OGRODJEM MEF

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  VIŠ. PRED. MAG. BOŠTJAN REŽMAH

  in somentorstvom (naziv, ime in priimek)

  _____

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v DKUM.

V Mariboru, dne 21.9.2011 Podpis avtorja/-ice: Matej Spindler

Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko

IZJAVA O USTREZNOSTI DIPLOMSKEGA DELA

Podpisani mentor BOŠTJAN KEŽMAH izjavljam, da je
(ime in priimek mentorja)
študent MATEJ ŠPINDLER izdelal diplomsko
(ime in priimek študenta-tke)

delo z naslovom: RAZVOJ NAMIZNIH APLIKACIJ V WPF Z ARHITEKTURNIM VZORCEM MVVM IN OGRODJEM MEF
(naslov diplomskega dela)

v skladu z odobreno temo diplomskega dela, Navodili o pripravi diplomskega dela in mojimi navodili.

Datum in kraj: 21.9.2011, MARIBOR

Podpis mentorja:

UNIVERZA V MARIBORU
Fakulteta za elektrotehniko, računalništvo in informatiko
(ime fakultete)

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE DIPLOMSKEGA DELA IN OBJAVI OSEBNIH PODATKOV AVTORJA

Ime in priimek avtorja (avtorice): Matej Spindler
Vpisna številka: E1008793
Študijski program: FERI - ITK VS RAZVOJ INFORMACIJSKIH SISTEMOV
Naslov diplomskega dela: RAZVOJ NAMIZNIH APLIKACIJ V WPF Z ARHITEKTURNIM VZORCEM MVVM IN OGRODJEM MEF

Mentor: Boštjan Kežmah
Somentor:

Podpisani-a Matej Spindler izjavljam, da sem za potrebe arhiviranja oddal-a elektronsko verzijo diplomskega dela v Digitalno knjižnico Univerze v Mariboru. Diplomsko delo sem izdelal-a sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah (Ur. l. RS, št. 16/2007) dovoljujem, da se zgoraj navedeno diplomsko delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija diplomskega dela je istovetna elektronski verziji, ki sem jo oddal-a za objavo v Digitalno knjižnico Univerze v Mariboru. Podpisani-a izjavljam, da dovoljujem objavo osebnih podatkov, vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum zagovora, naslov zaključnega dela) na spletnih straneh in v publikacijah UM.

Kraj in datum: Maribor, 20.09.2011

Podpis avtorja (avtorice):