Interakcija čovek - računar

Vežbe 6

Fakultet tehničkih nauka
Univerzitet u Novom Sadu

Napredni GUI dizajn – komande, stilovi, okidači

Komande

U okviru vežbi 3. *Uvod u WPF* objašenjen je osnovni način za obradu događaja (eng. *event handling*). Za svaku kontrolu može se izabrati na koje događaje ona reaguje, pri čemu je potrebno definisati reakciju, odnosno aktivnost koja se obavlja kada se događaj desi (eng. *event handler)*.

Međutim, za moderan korisnički interfejs, karakteristično je da se ista aktivnost može pokrenuti sa više mesta, različitim delovanjem korisnika. Na primer, kreiranje novog dokumenta obično se može pokrenuti izborom stavke menija, izborom ikonice na paleti alatki, pritiskom prečice na tasturi i izborom odgovarajuće opcije iz kontekstnog menija. Svi ovi događaji treba da pokrenu istu aktivnost, ali upotreba mehanizma obrade događaja zahetvala bi najmanje četiri obrađivača događaja (eng. *event handler*), što ne predstavlja idealno rešenje. Zbog toga *WPF* poseduje koncept komandi.

Komande odvajaju logiku pozivaoca od logike koja izvršava komandu. Pre svega, ovo razrešava prethodno opisan problem pokretanja iste aktivnost na više načina. Nadalje, logika se može prilagoditi objektu nad kojim se izvršava aktivnost. Na primer, aktivnost isecanja se uvek pokreće na isti način, ali sama logika isecanja slike i teksta ne mora uvek biti ista. Takođe, nije svaku aktivnost moguće uvek pokrenuti. Na primer, isecanje teksta nije moguće ako tekst nije selektovan i isto pravilo važi bez obzira na koji način se pokreće isecanje. Komande olakšavaju određivanje kada je neka aktivnost moguća, tako da pravilo nije potrebno zasebno definisati za svaki mogući način pokretanja aktivnosti.

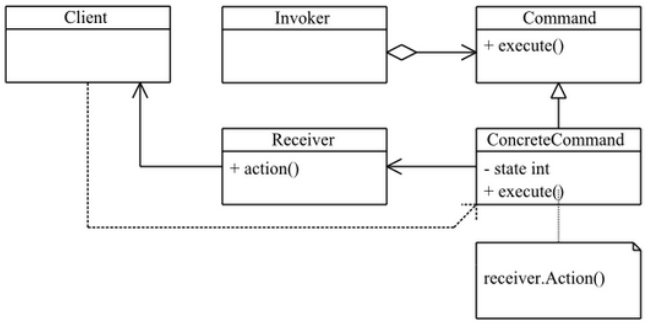
Osnovni pojmovi

Za upotrebu komandi u WPF-u potrebno je razlikovati četiri koncepta:

1. izvor komande (eng. *command source*) – predstavlja objekat koji pokreće izvršavanje komande,
2. komanda (eng. *command*) – predstvalja aktivnost koja će se izvršiti,
3. primaoc komande (eng. *command target*) – predstavlja objekat nad kojim se komanda izvršava,
4. veza logike i komande (eng. *command binding*) – mapiranje između komande i logike.

Izvori komande obično su obično kontrole poput dugmića i stavki menija. Komande su implementacije interfejsa *ICommand*. Interfejs *ICommand* sadrži metode *execute()* – logika izvršavanja i *canExecute()* – logika da li je komanda omogućena, kao i događaj *CanExecuteChanged* koji omogućava obaveštavanje kontrola o promenama statusa izvršivosti komande. Osnovna implementacija interfejsa *ICommand* je klasa *RoutedCommand*. Metode *execute()* i *canExecute()* klase *RoutedCommand* pokreću događaje koji prolaskom kroz stablo (eng. *bubbling and tuneling*) kontrola pronalaze onu kontrolu koja sadrži mapiranje između odgovarajuće komande i logike izvršavanja (eng. *command binding*). Komandu izvršava prva kontrola koja ima odgovarajući vezu. Komanda se izvršava nad primaocem komande.

WPF komande zasnovane su na upotrebi šablona ponašanja *Command Pattern* (slika 1). *Command Pattern* enkapsulira zahtev za obavljanje aktivnosti u objekat, pa na takav način obezbeđuje slabu spregnutost između logike pozivaoca i logike izvršioca aktivnosti, kao i jednostavno dodavanje novih akcija. Pored toga, šablon omogućava implementaciju redova čekanja za obavljanje akcija, logovanje akcija i obaveštavanje kada akcija nije moguća.



Slika 1. Command Pattern u opštem slučaju

Klasa *Invoker* pokreće realizaciju komande. U *WPF*-u ovo je sam korisnik koji komande pokreće upotrebom odgovarajućih kontrola poput dugmića i stavki menija. *Command* predstavlja generalizaciju svih komandi. U *WPF*-u ovo je interfejs *ICommand*. Klasa *ConcreteCommand* predstavlja konkretnu komandu. U *WPF*-u, ovo je ili ugrađena komanda ili nova komanda razvijena za potrebe aplikacije. Klasa *Reciever* je komponenta sposobna da primi i izvrši komandu. U WPF-u ona se specificira upotrebom veze komande i logike (eng. *command binding)*. Klasa *Client* predstavlja objekat nad kojim se izvršava komanda, odnosno primaoca komande (eng. *command target*).

WPF dolazi sa setom ugrađenih komandi. Specifične (eng. *custom commands*) komande mogu se napraviti implementacijom interfejsa *ICommand*. Međutim, češći pristup je da kreiraju upotrebom klasa *RoutedCommand* i *RoutedUICommand*. Oba načina prikazana su u primeru.

Ugrađene komande

WPF dolazi sa setom ugrađenih komandi. Ugrađene komande su podeljene u pet klasa:

1. *ApplicationCommands* – obuvata standardni set komandi klasičnih za sve aplikacije (*Copy, Paste, Cut, New, Open, Close, Delete, Find, Help, Print, Undo, Redo, Save, SaveAs*, ...)
2. *NavigationCommands* – obuhvata standardni set komandi za navigaciju, ukljujući navigaciju u pretraživači i navigaciju kroz dokument (*BrowseBack, BrowseForward, BrowseHome, FirstPage, LastPage, NextPage, Search, Zoom, Favorites*, ...),
3. *MediaCommands* – obuhvata standrdni set komandi za rukovanje multimedijalnim sadržajima (*Play, Pause, Record, Stop, NextTrack, MuteVolume, FastForward*, ...),
4. *EditingCommands* – obuhvata standardni set komandi za uređivanje (*AlignCenter, CorrectSpellingError, IncreaseFontSize, TabBackward, TabForward, ToggleBold, TooglеItalic*, ...) i
5. *ComponenteCommands* – obuhvata standardni set komandi vezanih za komponente (*ScrollPageUp, ScrollPageDown, MoveFocusBack, MoveFocusDown, Expand SelectionDown, ExpandSelectionLeft*...).

Ove komande predstavljaju instancu klase *RoutedCommand*, odnosno ne sadrže logiku. Sama logika određuje se povezivanjem sa kontrolom. Na primer, komanda *Close* se drugačuje izvršava na različitim kontrolama. Za deo kontrola nije implementirana logika ove komande, pa je na programeru da je implememtira. Međutim, veliki broj kontrola nudi implementaciju komandi sa kojima se najčešće povezuju. Na primer, *TextBox* kontrola nudi logiku za *Paste, Cut, Copy, Undo* i *Redo* komande.

Primer 1. Veza logike i komande (eng. command binding)

Za kontrolu prozor specificirane su veze između logike izvršavanja i komandi (slika 2). Dakle, komponenta prozor sada može da izvrši ove komande. Atribut *Command* specificira detalje komande. Atribut *CanExecute* je funkcija, odnosno logika određivanja kada se komanda može upotrebiti. Atribut *Executed* je funkcija, odnosno logika koja predstavlja samo izvršavanje komande.

```
<Window.CommandBindings>
    <CommandBinding Command="cmd:RoutedCommands.HelloWorld" CanExecute="HelloWorld_CanExecute" Executed="HelloWorld_Executed"></CommandBinding>
    <CommandBinding Command="cmd:RoutedCommands.Enable" CanExecute="Enable_CanExecute" Executed="Enable_Executed"></CommandBinding>
    <CommandBinding Command="cmd:RoutedCommands.Komanda" CanExecute="Komanda_CanExecute" Executed="Komanda_Executed"></CommandBinding>
    <CommandBinding Command="cmd:RoutedCommands.Ugradjene" CanExecute="Ugradjene_CanExecute" Executed="Ugradjene_Executed"></CommandBinding>
</Window.CommandBindings>
```

Slika 2. Veza logike i komande (eng. command binding) – datoteka MainWindow.xaml

Logika izvršavanja komande *HelloWorld* smeštena je u metodi *HelloWord_Execute* (slika 3), klase *MainWindow.xaml.cs*. Izvršavanje komande podrazumeva da se otvori dijalog sa ispisanom porukom „*Hello world!*“. Logika koja određuje kada je komandu moguće zadati smeštena je u metodi *HelloWord_CanExecute* (slika 3), klase *MainWindow.xaml.cs*. Trenutno, komandu je uvek moguće upotrebiti.

```
private void HelloWorld_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}

private void HelloWorld_Executed(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Hello world!");
}
```

Slika 3. Logika izvršavanja i dostupnosti HelloWorld komande - datoteka MainWindow.xaml.cs

Opis komande dat je u okviru klase *RoutedComand.cs*. Ova klasa sadrži više statičkih polja tipa *RoutedUICommand*. Ta polja predstavljaju opise komandi (slika 4). Prilikom kreiranja objekta klase *RoutedUICommand* prosleđuju se opis, naziv i tip komande. Dodatno može se proslediti lista ulaznih gestova koji pokreću komandu. Ulazni gestovi mogu biti pristici kombinacije tipki na tastaturi (prečice) ili određeni pokreti miša.

```
public static readonly RoutedUICommand HelloWorld = new RoutedUICommand(
    "Hello World",
    "HelloWorld",
    typeof(RoutedCommands),
    new InputGestureCollection()
    {
        new KeyGesture(Key.H, ModifierKeys.Control),
        new KeyGesture(Key.H, ModifierKeys.Alt | ModifierKeys.Control)
    }
    );
```

Slika 4. Opis HelloWorld komande

Ovako definisana komanda može se pokrenuti pomoću navedenih prečica. Međutim, u većini slučajeva pokretanje komande potrebno je omogućiti na još neki način. Na primer, potrebno je dodati stavku menija ili dugme koji pokreću komandu. Na slici 5. prikazana je stavka menija koja pokreće komandu *HelloWorld*. Donja crta u okviru Header atributa zabravo specificira mnemonik za stavku menija.

Mnemonici se aktiviraju klikom tastera ALT i predstavljaju još jedan alternativni način pokretanja komandi.

```
<MenuItem Header="_Hello World" Command="cmd:RoutedCommands.HelloWorld"></MenuItem>
```

Slika 5. Stavka menija za pokretanje HelloWorld komande

Dakle, komanda se može pokrenuti pritiskom neke od prečica, upotrebom mnemonika i izborom stavke menija. S obzirom da je tip komande *RoutedCommands*, kroz stablo se traži kontrola sa kojom je povezana logika izvršanja komande. U ovom primeru, ta komponenta je sam prozor. Kada je pronađena komponenta izvršava se specificirana funkcija. Na isti način specificirane su i ostale komande u okviru menija *Komande*.

Primer 2. Specifična (eng. custom) komanda implementacijom ICommand interfejsa

Na slici 6. dat je primer komande koje implementira interfejs *ICommand*. Komandu je uvek moguće aktivirati, pa metoda *CanExecute* uvek vraća true. U metodi *Execute* smeštena je logika operacije.

```
public class TestKomanda : ICommand //da hocem da radimo i sa precicama, trebalo bi da nasledom RoutedUICommand, recimo
{
    public bool CanExecute(object parameter)
    {
        return true;
    }

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        //ovde ide model-level obrada komande, tj. deo
        //koji ne radi sa interfejsom
        Console.WriteLine("Test");
        Console.Beep();
    }
}
```

Slika 6. Specifična (eng. custom) komanda implementacijom interfejsa ICommand

Primer 3. Upotreba ugrađenih komandi

Upotreba ugrađenih komandi demonstrirana je pomoću tekstualnog polja i komandi *Cut, Copy* i *Paste*. U datoteci *UgradjeneKomande.xaml* kreirana su tri dugmića i dodeljene su im ugrađene komande. Ciljna kontrola je tekstualno polje, koje već ima povezanu logiku izvršavanja komandi, kao i logiku određivanja kad je komandu moguće aktivirati.

```
<Button Grid.Column="0" Grid.Row="0" Command="ApplicationCommands.Cut" CommandTarget="{Binding ElementName=txtMain}">Cut</Button>
<Button Grid.Column="1" Grid.Row="0" Command="ApplicationCommands.Copy" CommandTarget="{Binding ElementName=txtMain}">Copy</Button>
<Button Grid.Column="2" Grid.Row="0" Command="ApplicationCommands.Paste" CommandTarget="{Binding ElementName=txtMain}">Paste</Button>
<TextBox Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="3" Name="txtMain" AcceptsReturn="True" HorizontalAlignment="Stretch" VerticalAlignment='
```

Slika 7. Upotreba ugrađenih komandi (datoteka UgradjeneKomande.xaml)

Više o komandama

[1] https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/commanding-overview

[2] https://www.wpf-tutorial.com/commands/introduction/

Stilovi

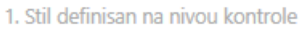
Stilovi predstavljaju skup opcija koji omogućavaju programerima i dizajnerima da lako kreiraju konzistentan i vizuelno primamljiv korisnički interfejs. Omogućavaju jednostavno održavanje izgleda, kao i deljenje istih vizuelnih efekata između više aplikacija. *WPF* poseduje stilove, koji funkcionišu slično kao *CSS* za *HTML*. Stil određuje vrednosti za skup osobina i dodeljuje se specifičnoj kontroli ili svim kontrolama nekog tipa. Stil može da nasledi drugi stil.

Primer 1. Definisanje stila na nivou kontrole

Stil se može definisati na nivou pojedinačne kontrole (slika 8). Stil definisan na ovaj način ima najveći prioritet. Dakle, ukoliko se osobina definiše na nivou komponente i na nivou svih kontrola određenog tipa, primenjivaće se vrednost specificirana na nivou kontrole. Na slici 9. Prikazan je izgleda labele.

```
<Label Content="1. Stil definisan na nivou kontrole">
    <Label.Style>
        <Style>
            <Setter Property="Label.FontSize" Value="10" />
            <Setter Property="Label.Foreground" Value="DarkGray" />
        </Style>
    </Label.Style>
</Label>
```

Slika 8. Stil definisan na nivou kontrole (datoteka StilPrimer.xaml)



Slika 9. Prikaz izgleda kontrole sa slike 8.

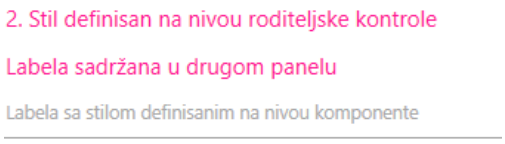
Primer 2. Definisanje stila na nivou roditeljske kontrole

Upotrebom *Resources* sekcije kontrole, može se definisati stil koji se primenjuje na kontrole sadžane u toj kontroli. Definisani stil primenjuje se na svim nivoima hijerarhije sadržavanja. Dakle, ako stil labele definišemo na nivou panela, zatim u njega smestimo naredni panel, pa tek u njega labelu, ona će i dalje poštovati definisani stil. Na isti način mogu se definisati stilovi na nivou prozora.

```
<StackPanel Grid.Row="1" Grid.Column="0">
    <StackPanel.Resources>
        <Style TargetType="Label">
            <Setter Property="Label.FontSize" Value="12" />
            <Setter Property="Label.Foreground" Value="DeepPink" />
        </Style>
    </StackPanel.Resources>
    <Label Content="2. Stil definisan na nivou roditeljske kontrole"></Label>
    <StackPanel>
        <Label Content="Labela sadržana u drugom panelu"></Label>
    </StackPanel>
    <Label Content="Labela sa stilom definisanim na nivou komponente">
        <Label.Style>
            <Style>
                <Setter Property="Label.FontSize" Value="10" />
                <Setter Property="Label.Foreground" Value="DarkGray" />
            </Style>
        </Label.Style>
    </Label>
    <Separator/>
</StackPanel>
```

Slika 10. Stil definisan na nivou roditeljse kontrole (datoteka StilPrimer.xaml)

Izgled komponenti prikazan je na slici 11.



Slika 11. Prikaz izgleda kontrola sa slike 10.

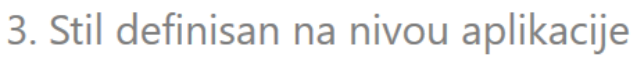
Primer 3. Stil na nivou aplikacije

Stil na nivou aplikacije treba da se primeni na kontrole u celoj aplikaciji, bez obzira o kom prozoru se radi. Ovi stilovi definišu se unutar datoteke *App.xaml* upotrebom *Resource* sekcije (slika 12).

```
<Application x:Class="NapredneKontrole.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
    <Style TargetType="Label">
        <Setter Property="Foreground" Value="Gray" />
        <Setter Property="FontSize" Value="24" />
    </Style>
    </Application.Resources>
</Application>
```

Slika 12. Stil definisan na nivou aplikacije (datoteka App.xaml)

Prikaz izgleda labele sa stilom definisanim na nivou aplikacije dat je na slici 13.



Slika 13. Labela sa stilom definisanim na nivou aplikacije

Primer 4. Eksplicitna upotreba stila

Svi do sada predstavljeni stilovi odnosili su se na izabrani tip kontrole i sve kontrole tog tipa su ih primenjivale. Međutim, u velikom broju slučajeva želimo da kontrole istog tipa na istom nivou hijerarhije koriste različite stilove. Na primer, želimo da se razlikuju labele koje su nazivi polja forme i labele koje su pomoćni tekst za polja. Ovakvo ponašanje može se postići postavljanjem atributa *x:Key* za stil (slika 13).

```
<Style x:Key="fancyButtonStyle" TargetType="Button">
    <Setter Property="Margin" Value="5"/>
    <Setter Property="Padding" Value="25"/>
    <Setter Property="BorderThickness" Value="0"/>
    <Setter Property="FontWeight" Value="Bold"/>
    <Setter Property="OpacityMask">
        <Setter.Value>
            <RadialGradientBrush GradientOrigin="0.5, 0.5" Center="0.5,0.5" RadiusX="0.5" RadiusY="0.5">
                <RadialGradientBrush.GradientStops>
                    <GradientStop Offset="0.2" Color="Black"></GradientStop>
                    <GradientStop Offset="0.75" Color="Transparent"></GradientStop>
                </RadialGradientBrush.GradientStops>
            </RadialGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

Slika 13. Stil sa postavljenim atributom x:Key

Kontrole koje žele da koriste imenovani stil moraju ga direktno specificirati (slika 14). Ostale kontrole tog tipa neće koristiti stil.

```
<StackPanel Grid.Row="3" Grid.Column="0">
    <Label Content="4. Imenovani stil"></Label>
    <Button>Bez stila</Button>
    <Button Style="{StaticResource ResourceKey=fancyButtonStyle}">Primenjen stil</Button>
    <Separator/>
</StackPanel>
```

Slika 14. Imenovani stil primenjen na kontroli

Na slici 15. prikazan je izgled dugmića bez primenjog stila i dugmića sa primenjenim stilom.

Slika 15. Izgled kontrole sa i bez primenjenog stila sa slike 13.

Primer 5. Okidači bazirani na osobinama kontrola

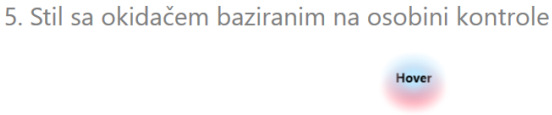
U prethodnim primerima, stilovi su zadavali vrednost osobinama statički. Upotrebom okidača, definisana vrednost osobine može se menjati u skladu sa ispunjenjem nekog uslova. Postoji više tipova okidača: okidači zasnovani na osobinama kontrola, okidači zasnovani na događajima i okidači zasnovani na podacima.

Okidači zasnovani na osobinama kontrola su najčešći tip okidača. Definišu se pomoću *<Trigger>* elementa. Okidač posmatra definisanu osobinu konrole i kada osobina dobije specificiranu vrednost, stil se aktivira. Na slici 16. prikazan je primer stila sa okidačem baziranim na osobini kontrole. Stil se aktivira kada se miš pozicionira na dugme (eng. *hover*). Kada osobina *IsMouseOver* ima vrednost *true*, stil je aktivan. U suprotnom stil nije aktivan.

```
<Style x:Key="fancyButtonHoverStyle" TargetType="Button"  BasedOn="{StaticResource ResourceKey=fancyButtonStyle}">
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Effect">
                <Setter.Value>
                    <DropShadowEffect Color="Lavender" Opacity="80" Direction="270" ShadowDepth="10"/>
                </Setter.Value>
            </Setter>
        </Trigger>
    </Style.Triggers>
</Style>
```

Slika 16. Stil sa okidačem baziranim na osobini kontrole

Na slici 17. prikazan je izgled kontrole na kojoj je prethodno prikazani stil aktiviran.



Slika 17. Kontrola sa aktivnim stilom sa okidačem

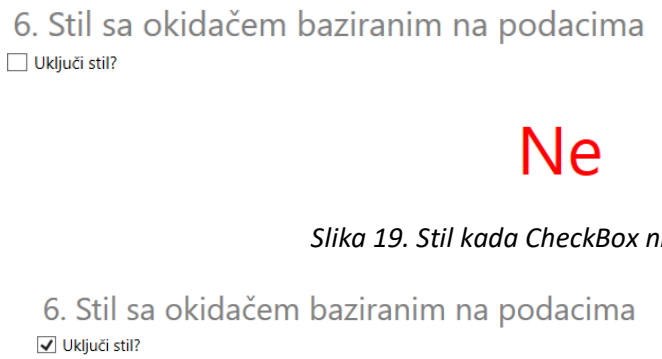
Primer 6. Stil sa okidačem baziranim na podacima

Okidači bazirani na podacima koriste se za povezivanje sa podacima koji nisu nužno osobine kontrole na koju se stil primenjuje. Definišu se pomoću *<DataTrigger>* elementa. Jedna od mogućih upotreba ovih okidača je vezivanje stila jedne kontrole za osobinu druge kontrole.

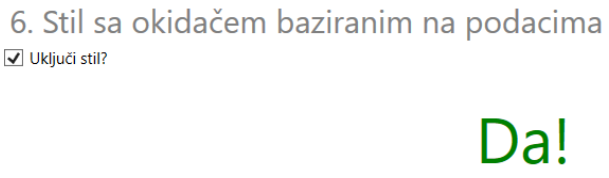
Na slici 18. dat je primer stila baziranog na podacima. *TextBlock* prikazuje crvenim slovima tekst ne, ukoliko nije označen *CheckBox* (slika 19). Ukoliko je označen *CheckBox*, *TextBlock* pokazuje crvenim slovima tekst da (slika 20).

```
<StackPanel Grid.Row="5" Grid.Column="0">
    <Label Content="6. Stil sa okidačem baziranim na podacima"></Label>
    <CheckBox Name="cbSample" Content="Uključi stil?" />
    <TextBlock HorizontalAlignment="Center" Margin="0,20,0,0" FontSize="48">
        <TextBlock.Style>
            <Style TargetType="TextBlock">
                <Setter Property="Text" Value="Ne" />
                <Setter Property="Foreground" Value="Red" />
                <Style.Triggers>
                    <DataTrigger Binding="{Binding ElementName=cbSample, Path=IsChecked}" Value="True">
                        <Setter Property="Text" Value="Da!" />
                        <Setter Property="Foreground" Value="Green" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </TextBlock.Style>
    </TextBlock>
    <Separator/>
</StackPanel>
```

Slika 18. Stil sa okidačem baziranim na podacima



Slika 19. Stil kada CheckBox nije označen



Slika 20. Stil kada CheckBox je označen

Primer 7. Stil sa okidačima baziranim na događajima

Okidači bazirani na događajima označavaju se elementom *<EventTrigger>.* Najčešće se koriste da bi pokrenuli animacije.

U primeru na slici 21. kreiran je stil koji se menja na osnovu događaja pomeranja miša na kontrolu *MouseEnter*, kao i na osnovu događaja pomeranja miša sa kontrole *MouseLeave*. Ideja je da se upotrebe animacije kako bi se tekst postepeno uvećavao pri incijalnom pozicioniranju miša sve dok ne dođe do izabrane veličine. Kada miš napusti kontrolu, tekst se postepeno smanjuje do izabrane veličine.

```
<StackPanel Grid.Row="6" Grid.Column="0">
    <Label Content="6. Stil sa okidačem baziranim na događaju"></Label>
    <TextBlock Name="lblStyled" Text="Pozicioniraj miš ovde!" FontSize="18" HorizontalAlignment="Center" VerticalAlignment="Center">
        <TextBlock.Style>
            <Style TargetType="TextBlock">
                <Style.Triggers>
                    <EventTrigger RoutedEvent="MouseEnter">
                        <EventTrigger.Actions>
                            <BeginStoryboard>
                                <Storyboard>
                                    <DoubleAnimation Duration="0:0:0.300" Storyboard.TargetProperty="FontSize" To="28" />
                                </Storyboard>
                            </BeginStoryboard>
                        </EventTrigger.Actions>
                    </EventTrigger>
                    <EventTrigger RoutedEvent="MouseLeave">
                        <EventTrigger.Actions>
                            <BeginStoryboard>
                                <Storyboard>
                                    <DoubleAnimation Duration="0:0:0.800" Storyboard.TargetProperty="FontSize" To="18" />
                                </Storyboard>
                            </BeginStoryboard>
                        </EventTrigger.Actions>
                    </EventTrigger>
                </Style.Triggers>
            </Style>
        </TextBlock.Style>
    </TextBlock>
</StackPanel>
```

Slika 21. Stil sa okidačem baziranim na događaju

Više o stilovima

[1] https://docs.microsoft.com/en-us/dotnet/desktop-wpf/fundamentals/styles-templates-overview

[2] https://www.wpf-tutorial.com/styles/introduction/

[3] https://www.wpf-tutorial.com/styles/trigger-datatrigger-event-trigger/