Лабораторная Работа № 7.
Шаблон MVVM, механизм привязок, команды WPF
Цель: Изучить механизм привязки данных в WPF.

Задание

1. Ознакомиться с краткими теоретическими сведениями.

2. Реализовать логику работы форм, реализованных в предыдущей лабораторной работе, при помощи шаблона MVVM с использованием привязок по данным и по командам.

3. Оформить отчет о выполнении лабораторной работы.

Краткие теоретические сведения

Механизм привязок WPF

Механизм привязки данных в WPF предоставляет простой способ связывания визуального интерфейса и данных в виде объектов .NET. При этом, в случае правильной настройки привязки и реализации механизма нотификации (посредством интерфейса INotifyPropertyChanged) при изменении данных в визуальном интерфейсе (например, заполнение TextBox'a) привязанный класс данных тоже изменяется. Рассмотрим пример:

```
<TextBlock x:Name="horseName" Text="{Binding Path=Name}" />
public class Horse
{
public string Name { get;set;}
}
public partial class AWindow : Window
   {
      public AWindow()
      {
         InitializeComponent();
this.DataContext = new Horse { Name = «MyHorse» };
      }
   }
```

Механизм привязок работает следующим образом. Определяется класс данных, содержащий значение — в данном случае это Horse. На желаемое свойство элемента управления создается привязка и указывается, к какому свойству привязываться (Path = «Name»). TextBox знает, что необходимо обращаться к классу Horse, потому что он установлен как DataContext на окне. DataContext — это объект, связанные с элементом управления, по которому берется значение по привязке. В случае, если он не установлен — берется DataContext первого родительского элемента, у которого он установлен (в данном случае — окно).

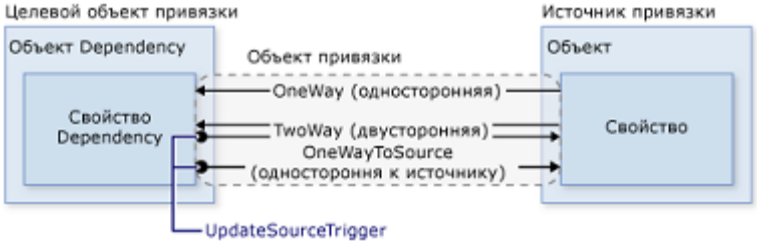
В результате при изменении данных в окне, в экземпляре класса Horse значение Name также изменится — автоматически, без дополнительного кода подвязки. Чтоб заработала обратная сторона интеграции — изменения в экземпляре класса Horse были видны в визуальном интерфейсе — необходимо реализовать интерфейс InotifyPropertyChanged:

```
public class Horse:INotifyPropertyChanged
{
   private string _name;
   #region INotifyPropertyChanged Members
   public event PropertyChangedEventHandler PropertyChanged;
   private void OnPropertyChanged(string propertyName)
   {
      if (PropertyChanged != null)
      {
         PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
      }
```

```
    }
    #endregion
    public string Name
    {
        get{ return _name; }
        set
        {
            _name = value;
            OnPropertyChanged("Name");
        }
    }
}
```

Механизм привязок проверяет реализацию данного интерфейса и в случае успеха подписывается на событие изменения свойства. При всяком изменении свойства Name автоматически вызывается событие PropertyChanged, и механизм привязок считывает значение и устанавливает его в визуальном элементе. Привязки также имеют некоторые настройки. По обновлению взаимодействующих сторон они бывают:

- Двусторонние (обновляется и визуальный интерфейс, и класс данных);
- Односторонние к визуальному интерфейсу;
- Односторонние к классу данных;
- Односторонние к визуальному интерфейсу 1 раз;



По способу получения данных можно привязки организовывать следующим образом:

- Из контекста данных (по-умолчанию) — через DataContext;
- Из свойств других элементов управления — при помощи ElementName;
- Из свойств других элементов — при помощи относительной привязки (RelativeSource);

Привязки можно также использовать вместе с триггерами. Например, можно настроить триггер данных, который будет изменять рамку кнопки в зависимости от значения в модели. Также настраивается момент обновления источника/приемника привязки(UpdateSourceTrigger):

- LostFocus — значение в модели обновляется по потере фокуса элемента управления;
- PropertyChanged — при вводе (при нажатии каждой клавиши);
- Explicit — когда должно вызываться из программного кода.

Некоторые примеры привязок:

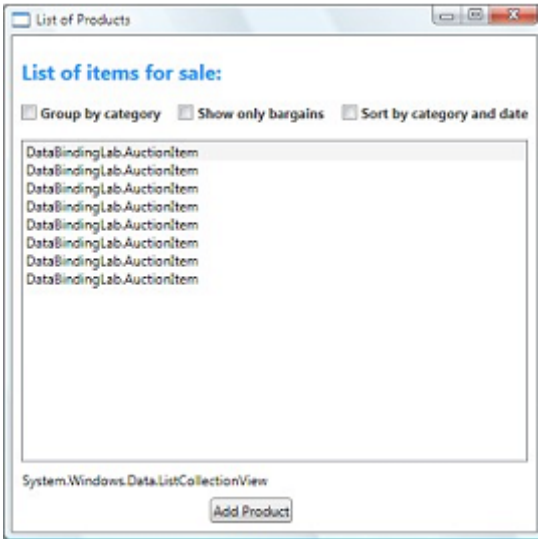
```
<DockPanel.Resources>
  <c:MyData x:Key="myDataSource"/>
</DockPanel.Resources>
<Button Width="150" Height="30"
      Background="{Binding Source={StaticResource myDataSource},
                  Path=ColorName}">I am bound to be RED!</Button>
<TextBox Name="StartPriceEntryForm" Grid.Row="2" Grid.Column="1"
   Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5">
  <TextBox.Text>
    <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">
      <Binding.ValidationRules>
        <ExceptionValidationRule />
      </Binding.ValidationRules>
```

```
    </Binding>
  </TextBox.Text>
</TextBox>
{Binding  Path=PathToProperty, RelativeSource={RelativeSource AncestorType={x:Type
typeOfAncestor}}}
```

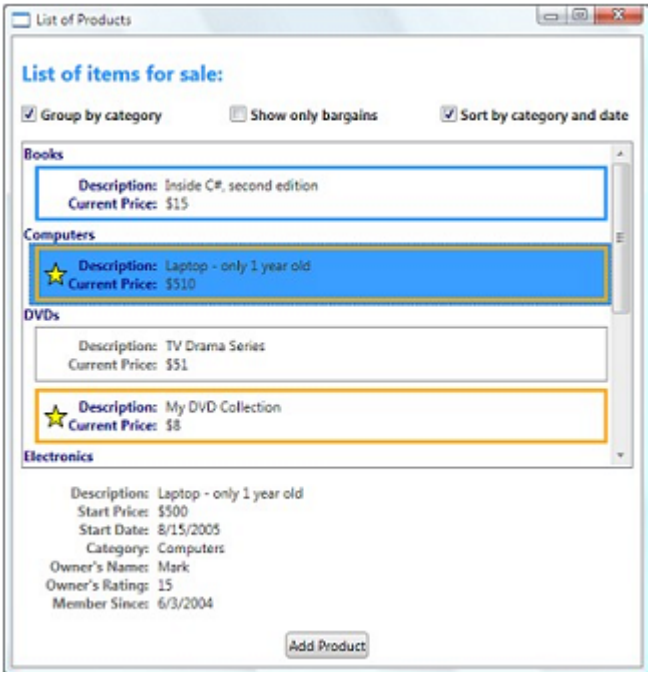
Шаблоны данных в WPF

Очень часто становится задача отобразить список объектов — например, список пациентов в больнице. При этом список пациентов — это набор .NET объектов — экземпляров класса Patient. Для красивого их отображения в представлении в WPF существует шаблоны данных — DataTemplate. Они указывают, какой внешний вид будут принимать объекты конкретного типа.

Рассмотрим пример.



Без использования шаблонов данных, привязанные объекты AuctionItem будут отображаться в виде TextBlock-ов с текстом, определенным в методе ToString() объектов. В случае, если мы хотим отобразить их следующим образом:



Достаточно добавить следующий шаблон данных:

```
<DataTemplate DataType="{x:Type src:AuctionItem}">
  <Border BorderThickness="1" BorderBrush="Gray"
      Padding="7" Name="border" Margin="3" Width="500">
```

```
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="20"/>
        <ColumnDefinition Width="86"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
        <Polygon Grid.Row="0" Grid.Column="0" Grid.RowSpan="4"
              Fill="Yellow" Stroke="Black" StrokeThickness="1"
              StrokeLineJoin="Round" Width="20" Height="20"
              Stretch="Fill"
              Points="9,2 11,7 17,7 12,10 14,15 9,12 4,15 6,10 1,7 7,7"
              Visibility="Hidden" Name="star"/>
        <TextBlock Grid.Row="0" Grid.Column="1" Margin="0,0,8,0"
               Name="descriptionTitle"
               Style="{StaticResource smallTitleStyle}">Description:</TextBlock>
        <TextBlock Name="DescriptionDTDataType" Grid.Row="0" Grid.Column="2"
           Text="{Binding Path=Description}"
           Style="{StaticResource textStyleTextBlock}"/>
        <TextBlock Grid.Row="1" Grid.Column="1" Margin="0,0,8,0"
               Name="currentPriceTitle"
               Style="{StaticResource smallTitleStyle}">Current Price:</TextBlock>
        <StackPanel Grid.Row="1" Grid.Column="2" Orientation="Horizontal">
          <TextBlock Text="$" Style="{StaticResource textStyleTextBlock}"/>
          <TextBlock Name="CurrentPriceDTDataType"
            Text="{Binding Path=CurrentPrice}"
            Style="{StaticResource textStyleTextBlock}"/>
        </StackPanel>
    </Grid>
</Border>
<DataTemplate.Triggers>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
      <DataTrigger.Value>
        <src:SpecialFeatures>Color</src:SpecialFeatures>
      </DataTrigger.Value>
      <DataTrigger.Setters>
        <Setter Property="BorderBrush" Value="DodgerBlue" TargetName="border" />
        <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
        <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
        <Setter Property="BorderThickness" Value="3" TargetName="border" />
        <Setter Property="Padding" Value="5" TargetName="border" />
      </DataTrigger.Setters>
    </DataTrigger>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
      <DataTrigger.Value>
        <src:SpecialFeatures>Highlight</src:SpecialFeatures>
      </DataTrigger.Value>
```

```
            <Setter Property="BorderBrush" Value="Orange" TargetName="border" />
            <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
            <Setter Property="Visibility" Value="Visible" TargetName="star" />
            <Setter Property="BorderThickness" Value="3" TargetName="border" />
            <Setter Property="Padding" Value="5" TargetName="border" />
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>
```

У шаблона данных (DataTemplate) обязательно указывается тип данных, к которому он относится. Шаблоны данных находятся в ресурсах, и могут также иметь ключ — в таком случае, они будут подключаться только если принудительно указать ключ к шаблону данных. Без указания ключа (x:Key) они будут установлены как шаблоны по-умолчанию.

Важно отметить, что шаблон данных может иметь триггеры — изменять внешний вид в зависимости от определенных значений либо экземпляра класса либо собственных свойств (наведение мышки и др.). Шаблон данных во многом напоминает шаблон элемента управления — только вместо элемента управления стилизуются обыкновенные объекты, которые поставлены в качестве DataContext у элементов управления в свойство Content.

Механизм команд WPF

Команда в WPF — это действие, которое должно быть выполнено при каком-то событии. Команды заменяют собой событийную модель реализации логики представления, поскольку вместо подписки на событие — к элементу управления привязывается объект команды, у которого будет вызван соответствующий метод. Все команды реализуют интерфейс ICommand:

```
public interface ICommand
{
public bool CanExecute(object parameter);
public void Execute(object parameter);
event EventHandler CanExecuteChanged;
}
```

При выполнении какого-то события (например, щелчок на кнопку), у команды вызывается метод Execute с параметром, который указан дополнительно в привязке. Метод CanExecute используется для отключения элементов управления (IsEnabled). Простейший пример привязки команды:

```
<Button Content="Click"
        Height="23"
        HorizontalAlignment="Left"
        Margin="77,45,0,0"
        Name="btnClick"
        VerticalAlignment="Top"
        Width="203"
        Command="{Binding ButtonCommand}"
        CommandParameter="Hai" />
```

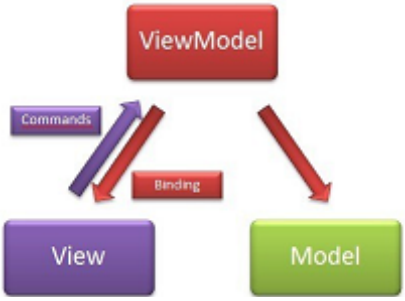
Параметр команды может браться из класса данных, связанных с элементом управления при помощи привязки. Команды предоставляют простой способ привязки функций к соответствующим событиям элементов управления. Шаблон MVVM

Шаблон Model-View-ViewModel (MVVM) применяется для разделения ответственности между компонентами программного обеспечения, облегчения разработки и поддержки приложений с насыщенной логикой визуального интерфейса, повышения тестируемости кода и другое. Данный шаблон получил широкое применение в приложения WPF ввиду простоты его внедрения и удобства использования. Основная проблема при написании приложений WinForms или WPF с применением событийной системы реализации бизнес-логики работы — огромные массы кода в code-behind классах представлений (форм, view), которые сложно

отслеживать и поддержива. Например, когда необходимо предпринимать некоторые действия при изменении галочки (снята/добавлена), приходится подписываться на множество событий, при этом не всегда тривиально и ясно, каким образом конкретно протекает ход событий. Также сложна логика заполнения формы значениями — как правило это выделенные методы InitForm, ClearForm(для очистки всех текстовых полей и галочек от значений) и прочее. В замену этим трудностях рекомендуется использовать специальные классы, содержащие в себе полностью всю логику отработки представления — логика работы кнопок, наведения мышки и прочее. Этот же класс должен хранить состояния галочек, введенные текстовые значения и другое. Такой класс называется модель представления (view model) — он хранит все данные и логику представления (view). Необходимо сразу отметить, что он не имеет никакой логики, касающейся отрисовки — вся она должна остаться в представлении (view).

Третий элемент шаблона — модель — это чистые классы данных системы либо сервисы или слой доступа к данным, из которого можно получить необходимые данные для заполнения формы. Например, если это форма редактирования товара, то моделью тут будет товар, его описание, связанные товары и прочее. Модель представления (viewmodel) использует сервисы и данные, получаемые от модели (model) для заполнения представления (view). Для достижения этой цели viewmodel использует механизм привязок и специальный интерфейс

INotifyPropertyChanged.



Таким образом, для каждого представления создается специальный класс, содержащий всю логику работы представления без деталей визуального отображения. И механизм привязок (команд и свойств) позволяет легко и без усилий перенести всю эту логику на визуальный интерфейс.

Контрольные вопросы

3. Что такое привязки в WPF ?
4. Как осуществляется привязка команд в WPF ?
5. Что такое MVVM ?