



Разработка приложений
с использованием WPF

Урок № 4

Расширенные
приемы работы
с элементами
управления

Содержание

1. Локализация приложений	
при помощи файлов ресурсов	5
1.1. Добавление локализации в проект Visual Studio.....	10
1.1.1. Терминология и сокращения.....	21
2. Расширения разметки.....	22
2.1. Создание собственных расширений разметки.....	23
2.1.1. Пространства имен.....	24
2.1.2. Интерпретация текста при использовании расширений разметки.....	26
2.1.3. Пример 1	27
2.1.4. Пример 2	29
2.2. Стандартные расширения разметки XAML	31
2.2.1. Расширение разметки x:Type.....	32
2.2.2. Расширение разметки x:Null.....	33

2.2.3. Расширение разметки x:Static	34
2.2.4. Расширение разметки x:Array	36
2.3. Стандартные расширения разметки WPF	38
2.4. Привязка данных.....	38
2.4.1. Направление потока данных	40
2.4.2. Расширение разметки Binding.....	42
2.4.3. Триггеры обновлений источника данных	51
2.4.4. Привязка коллекций.....	55
3. Шаблоны данных.....	59
4. Команды	66
4.1. Создание команд	67
4.1.1. Реализация интерфейса ICommand.....	68
4.2. Горячие клавиши	78
4.3. Команды для событий.....	79
5. Архитектурный шаблон проектирования MVVM	85
5.1. Введение в архитектуру Model-View-X	85
5.1.1. Описание архитектурного шаблона проектирования MVC	86
5.1.2. Описание архитектурного шаблона проектирования MVVM	87
5.2. Обоснование использования архитектурного шаблона проектирования MVVM	91
5.2.1. Упрощенная архитектура приложения.....	92
5.2.2. Упрощенная архитектура приложения (WPF вариант).....	93

5.2.3. Проблемы упрощенной архитектуры приложения.....	94
5.2.4. MVVM архитектура приложения	96
5.3. Достоинства применения архитектурного шаблона проектирования MVVM.....	96
5.4. Процесс внедрения MVVM в существующее приложение	97
5.4.1. Вариант 1. Начальный.....	99
5.4.2. Вариант 2. Задействование особенностей WPF	100
5.4.3. Вариант 3. Применение архитектурного шаблона MVVM	100
5.4.4. Вариант 4. Выделение общего класса для Моделей Представления.....	101
6. Домашнее задание	102

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Локализация приложений при помощи файлов ресурсов

При разработке приложений, соответствующих сегодняшним стандартам индустрии, одним из очень важных этапов является локализация. Локализация — это процесс адаптации программного продукта к языку и культуре целевой аудитории (пользователя приложения). Вот некоторые из основных аспектов, которые учитываются при этом процессе:

- Перевод пользовательского интерфейса, который может включать помимо строк еще и изображения и прочий контент.
- Перевод документации, например, справки поставляемой вместе с приложением.
- Контроль формата даты, времени и денежных единиц.
- Учет возможной раскладки клавиатуры.
- Использование корректной символики и цветовой гаммы.

Совместно с локализацией часто используется другой термин — интернационализация, который очень часто путают с первым. Интернационализация — это более обобщенный термин, который подразумевает проектирование и разработку программного продукта таким образом, который максимально упростит процесс локализации.

Сюда можно отнести такие моменты, как выбор кодировки символов (например, Unicode), использование файлов ресурсов, которые будут рассмотрены в этом разделе или выбор WPF в качестве технологии разработки из-за ее системы автоматического вычисление размеров элементов, что позволит «тянуть» пользовательский интерфейс под разные размеры, в зависимости от языка.

Локализация играет очень важную роль на сегодняшний день в сфере разработки программного обеспечения. Существует много различных мест в приложении, требующих локализации. Наиболее важную роль играет локализация пользовательского интерфейса (рис. 1), так как именно с этой частью приложения пользователь взаимодействует напрямую. Помимо пользовательского интерфейса также можно упомянуть и другие составляющие программного продукта, требующие локализации. Например, приложение может генерировать отчет о произведенных финансовых операциях в виде текстового файла. Некоторые приложения содержат что-то наподобие встроенных уроков для обучения работе с ним. Такие уроки могут сопровождаться голосовым озвучиванием, которое может быть разным, и зависеть от региональных настроек пользовательского окружения. Существует и ряд других ситуаций, в которых локализация станет уместной и даже необходимой.

Реализовать локализацию в приложение можно не одним способом. Все зависит от того, насколько сложные условия накладываются на алгоритмы, обрабатывающие локализуемые элементы. Например, если речь идет о строках, которые отображаются на элементах управления

1. Локализация приложений при помощи файлов...

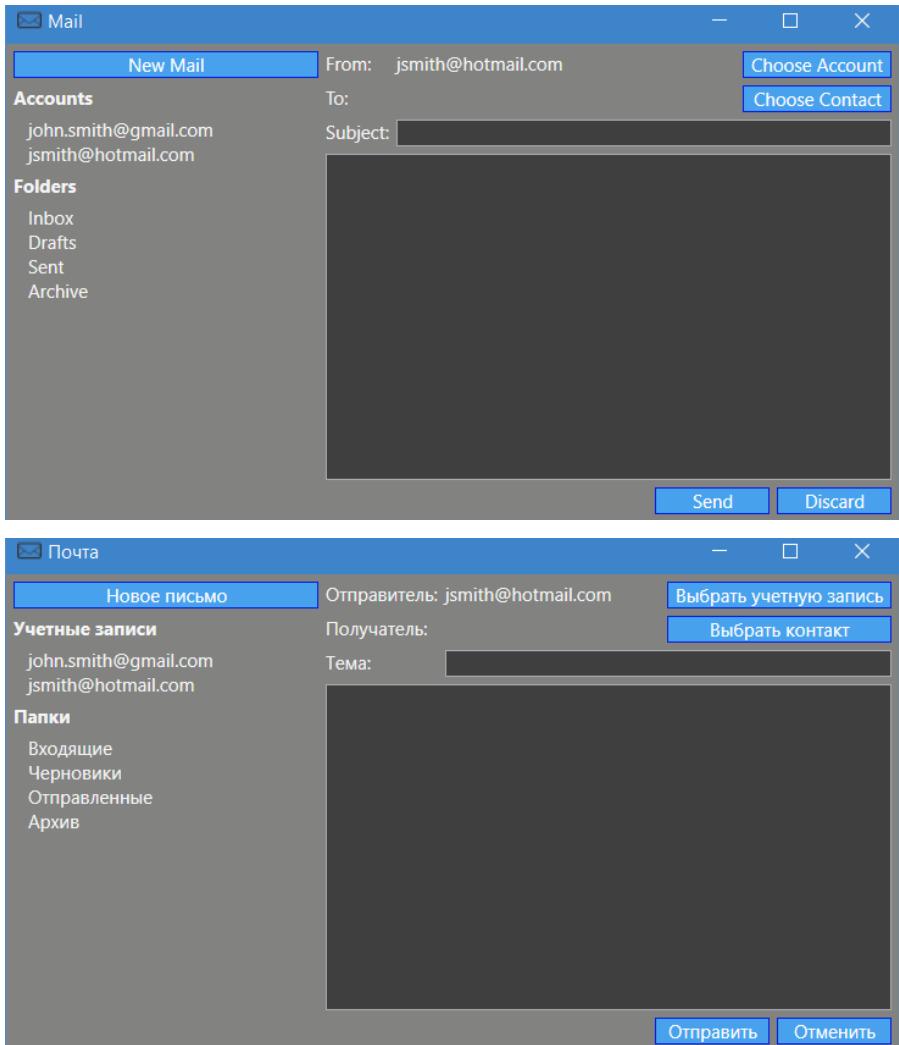


Рис. 1. Локализация пользовательского интерфейса

пользовательского интерфейса, то реализация может усложниться, если строки необходимо генерировать динамически, или же наоборот, упроститься, если строки, в основном, фиксированные.

В данном разделе будет рассмотрен стандартный способ локализации, заложенный в .NET Framework. Это означает, что применим он не только к WPF-приложениям, но и ко всем .NET-приложениям. Этот способ основывается на создании нескольких файлов ресурсов, количество которых зависит от числа поддерживаемых вариантов локализаций. В один из файлов (основной) помещаются все ресурсы, которые подвержены локализации, такие как: текст, изображения, звуковые файлы и т.д. При этом, каждой единице ресурсов дается уникальный идентификатор. Далее создается по одномуциальному отдельному файлу на каждый вариант локализации, в которые помещаются аналогичные ресурсы, с такими же идентификаторами, чтобы получить соответствие оригинальному ресурсу, но в локализованной форме.

Visual Studio автоматически генерирует классы, в которые помещает свойства, через которые будет производиться доступ к ресурсам. В качестве названий данных свойств выступают упомянутые ранее уникальные идентификаторы ресурсов. После этого остается всего лишь использовать обращение к нужному из них в программном коде. Для программиста существует сразу несколько положительных моментов от такой структуризации. Во-первых, данная схема позволяет избежать дублирования ресурсов (например, повторения схожих строк в разных фрагментах программного кода). Во-вторых, все ресурсы получаются изолированными и их становится легче поддерживать, т.е. добавлять новые, удалять ненужные и обновлять устаревшие. На рис. 2 показан пример файла ресурса, содержащего основную версию локализации приложения (английскую).

1. Локализация приложений при помощи файлов...

The screenshot shows the Visual Studio Resource Editor interface. The title bar displays "Strings.resx" and "Strings.ru.resx". The toolbar includes "Add Resource" and "Remove Resource" buttons, along with an "Access Modifier: Public" dropdown. The main area is a table with columns "Name", "Value", and "Comment". The table contains the following data:

Name	Value	Comment
Accounts	Accounts	
Folder_Archive	Archive	
Folder_Drafts	Drafts	
Folder_Inbox	Inbox	
Folder_Sent	Sent	
Folders	Folders	
NewMail	New Mail	
NewMail_ChooseAccount	Choose Account	
NewMail_ChooseContact	Choose Contact	
NewMail_Discard	Discard	
NewMail_From	From:	
NewMail_Send	Send	
NewMail_Subject	Subject:	
NewMail_To	To:	
Window_Title	Mail	

Рис. 2. Файл ресурсов, содержащий основную версию локализации

На рис. 3 показан пример файла ресурсов, который содержит локализованный вариант строк для русского языка. Как можно заметить, колонка **Name** содержит те же самые идентификаторы, что и файл ресурсов с основной версией локализации, а колонка **Value** содержит перед соответствующими строками.

The screenshot shows the Visual Studio Resource Editor interface. The title bar displays "Strings.resx" and "Strings.ru.resx". The toolbar includes "Add Resource" and "Remove Resource" buttons, along with an "Access Modifier: Public" dropdown. The main area is a table with columns "Name", "Value", and "Comment". The table contains the following data:

Name	Value	Comment
Accounts	Учетные записи	
Folder_Archive	Архив	
Folder_Drafts	Черновики	
Folder_Inbox	Входящие	
Folder_Sent	Отправленные	
Folders	Папки	
NewMail	Новое письмо	
NewMail_ChooseAccount	Выбрать учетную запись	
NewMail_ChooseContact	Выбрать контакт	
NewMail_Discard	Отменить	
NewMail_From	Отправитель:	
NewMail_Send	Отправить	
NewMail_Subject	Тема:	
NewMail_To	Получатель:	
Window_Title	Почта	

Рис. 3. Файл ресурсов, содержащий дополнительный вариант локализации

1.1. Добавление локализации в проект Visual Studio

Ранее было сказано, что рассматриваемый процесс локализации идентичен для любого .NET-проекта и это так, поэтому весь процесс добавления локализации будет рассмотрен на примере WPF-проекта. Описанные далее шаги можно выполнять как в новом, только что созданном проекте, так и в уже существующем. Процесс создания самого проекта будет опущен.

Весь процесс внедрения локализации в проект сводится к добавлению нескольких файлов ресурсов (по одному файлу на каждую версию локализации). Среди этих файлов ресурсов всегда есть один основной, который содержит базовую версию локализации, а все остальные содержат различные варианты его переводов.

В первую очередь необходимо добавить в проект основной (главный) файл ресурсов. Для этого необходимо выбрать пункт меню **Project → Add New Item...**, или нажать комбинацию клавиш **Ctrl+Shift+A** (рис. 4).

В открывшемся окне, слева, необходимо выбрать **Visual C# → General**, после чего выбрать тип файла **Resources File**. В нижней части окна, как и при добавлении других типов файлов, находится поле **Name**, которое следует изменить на более подходящее, чем то, что написано изначально в качестве «заглушки» (рис. 5).

После добавления файла ресурсов в проект, сразу же откроется редактор ресурсов для этого файла (рис. 6). То же самое произойдет если попытаться открыть добавленный файл через окно **Solution Explorer** двойным щелчком левой кнопки мыши.

1. Локализация приложений при помощи файлов...

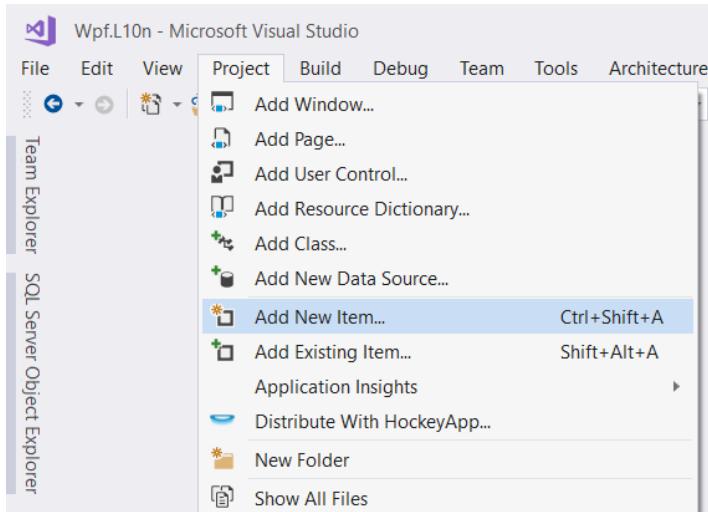


Рис. 4. Выбор пункта главного меню для добавления нового файла ресурсов

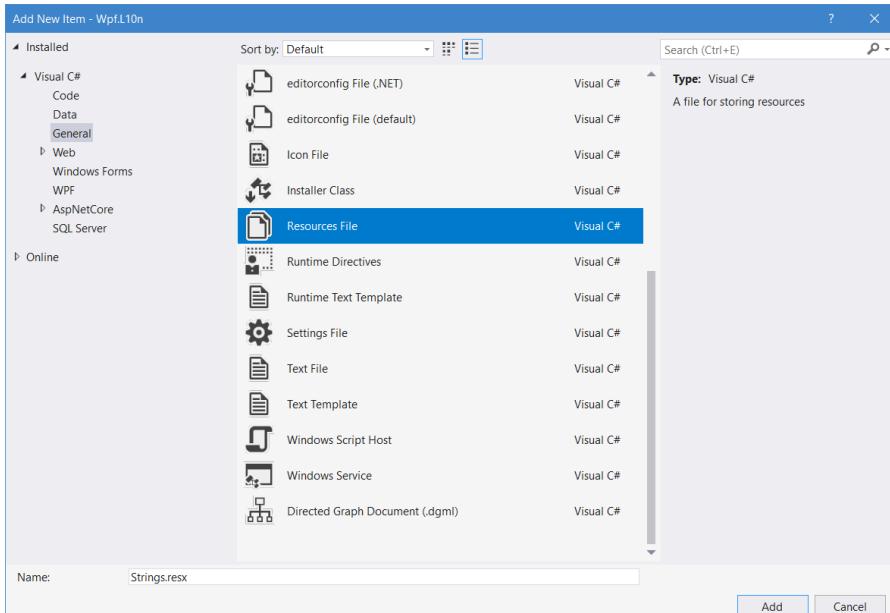


Рис. 5. Выбор типа файла

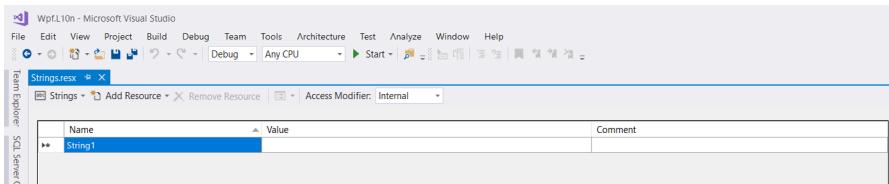


Рис. 6. Редактор ресурсов

Каждый файл ресурсов может содержать различные виды ресурсов. Данный пример будет сконцентрирован на строках, редактор которых изначально и открывается. Данный редактор содержит три колонки, с которыми придется взаимодействовать, чтобы добавлять новые строки в ресурсы. Колонка **Name** содержит уникальный идентификатор строки, который будет преобразован в .NET-свойство, следовательно на него накладываются все правила именования идентификаторов. Колонка **Value** содержит саму строку, которую необходимо отобразить на экран, сохранить в файл или сделать что-либо еще, т.е. это и есть тот ресурс ради которого все это делается. В пределах одного файла ресурсов все ресурсы должны относиться к одному и тому же варианту локализации, например английский или испанский язык. Колонка **Comment** может содержать дополнительный комментарий к строке, в котором можно указать, к примеру, где именно в интерфейса данная строка применяется. Первые два поля (**Name** и **Value**) любого добавленного ресурса являются обязательными к заполнению, последнее (**Comment**) — нет (рис 7).

Данный подход хорош еще и тем, что на текущем этапе уже можно использовать добавленные строки в приложении, даже если еще нет локализованных вариантов для других языков. Это дает возможность вести разработку

параллельно с локализацией приложения или же добавлять локализации позже.

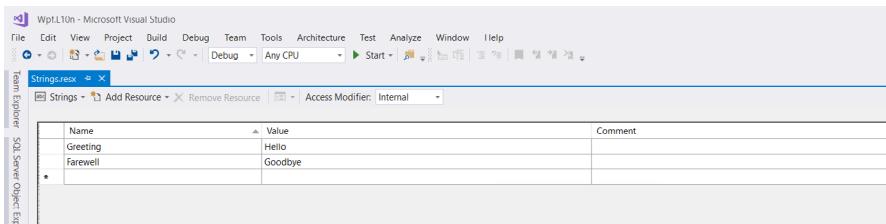


Рис. 7. Пример добавления строк в файл ресурсов

Даже если приложение на начальном этапе проектирования и разработки не требует локализации, внедрение данной концепции в него позволит сделать «заготовку» на будущее, если вдруг требования изменятся. При этом не придется переписывать уйму программного кода, а понадобиться всего лишь добавить новые варианты файлов ресурсов.

Для демонстрации использования созданных строк в пользовательском интерфейсе необходимо создать небольшой фрагмент XAML-разметки (полный пример находится в папке Wpf.L10n):

XAML

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition Width="5"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Button x:Name="greetingButton" Grid.Column="0"/>
    <Button x:Name="farewellButton" Grid.Column="2"/>
</Grid>
```

Далее в соответствующем файле с программным кодом (модель code-behind) необходимо написать следующий фрагмент программного кода:

C#

```
greetingButton.Content = Strings.Greeting;  
farewellButton.Content = Strings.Farewell;
```

Здесь происходит присваивание ресурсов согласно их идентификаторам. Все свойства являются статическими и помещаются в класс, который называется так же, как и файл ресурсов, т.е. Wpf.L10n.Strings в данном случае.

Результат приведенной выше разметки и программного кода показан на рис. 8.



Рис. 8. Использование строк из файла ресурсов

В данном примере используется программное присваивание значений свойствам элементов управления, но есть возможность обойтись только XAML-разметкой, используя специальные расширения разметки, о которых речь пойдет дальше.

Для того, чтобы добавить новый вариант локализации, например, для французского языка, необходимо добавить еще один файл ресурсов. Новый файл должен называться так же само, но при этом обладать еще и специальным суффиксом, в котором будет указано для какого именно языка (или региона) предназначен данный файл. Пол-

ный список кодов можно посмотреть здесь: <https://msdn.microsoft.com/en-us/library/hh441729.aspx?f=255&MSPError=-2147217396>.

Все коды форматируются одним из двух вариантов:

- **Указание языка.** Французский (fr), испанский (es), английский (en), русский (ru) и т.д.
- **Указание языка и региона (страны).** Американский английский (en-US), британский английский (en-UK).

Отличие двух форм заключается в том, что в одной из них указывается регион в дополнение к языку, и именно эта форма считается более приоритетной при определений какой файл с ресурсами выбирать. Например, у пользователя предпочтительной локализацией является канадская версия французского языка (fr-CA). Если такой версии файла не нашлось, то производится поиск файла ресурсов на соответствие только языку (fr), и, если такой присутствует, то будет использоваться он. Иначе берется основной файл ресурсов (без суффикса с кодом).

Согласно данному примеру, необходимо добавить новый файл ресурсов с названием Strings.fr.resx (рис. 9). Здесь .resx стандартное расширение для файлов данного типа.

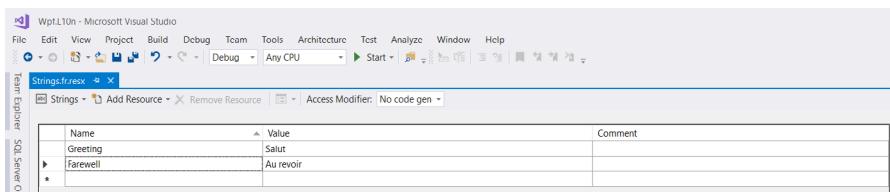


Рис. 9. Пример добавления локализованных строк в файл ресурсов

При запуске приложения будет выбран один из доступных файлов ресурсов автоматически. Какой именно язык является предпочтительным для пользователя определяется настройками операционной системы, а именно тем, какой язык установлен в качестве языка по умолчанию для отображения приложений. Однако, есть возможность явно указать, какую версию локализации стоит использовать. Для этого необходимо указать следующий фрагмент программного кода где-то в приложении, но до того, как начнется считывание файлов ресурсов.

C#

```
Thread.CurrentCulture =  
    new CultureInfo("fr");
```

Теперь независимо от настроек операционной системы пользователя, будет использована французская версия локализации.

Результат приведенного выше программного кода показан на рис. 10.



Рис. 10. Использование локализованных строк из файла ресурсов

Изменение свойства `System.Windows.Threading.Thread.CurrentCulture` часто производится как можно ближе к главной функции приложения. Связано это с тем, что значение этого свойства учитывается при

каждом считывании ресурса из файла с ресурсами, т.е. каждое обращение в программном коде к идентификатору ресурса, сначала определяет какая версия локализации требуется, после этого происходит поиск необходимого файла с ресурсами и собственно извлечение ресурса. Такое процесс происходит каждый раз, когда создаются элементы управления, для свойств которых применяются ресурсы и происходит это в методе InitializeComponent. Если изменить значение свойства текущей локализации на другое после этого, то элементы управления не обновятся, так как не происходит никакого события, на которое они могли бы подписаться. Однако, все последующие обращения к ресурсам будут возвращать последний установленный вариант локализации.

Для того, чтобы реализовать переключение языка «на лету», например, при нажатии на кнопку, придется вручную обновить значения всех элементов управления, использующих ресурсы. Данный процесс конечно же можно автоматизировать, но стандартного готового механизма для этого нет.

Приведенный ниже фрагмент разметки демонстрирует переключение локализации во время работы приложения (полный пример находится в папке Wpf.L10n. LanguageSwitch):

XAML

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="5"/>
```

```
<ColumnDefinition/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="5"/>
    <RowDefinition/>
    <RowDefinition Height="5"/>
    <RowDefinition/>
    <RowDefinition Height="5"/>
    <RowDefinition/>
    <RowDefinition Height="30"/>
    <RowDefinition/>
</Grid.RowDefinitions>
<TextBlock x:Name="nameTextBlock"
           Grid.Column="0" Grid.Row="0"/>
<TextBox Grid.Column="2" Grid.Row="0"/>
<TextBlock x:Name="surnameTextBlock"
           Grid.Column="0" Grid.Row="2"/>
<TextBox Grid.Column="2" Grid.Row="2"/>
<TextBlock x:Name="phoneTextBlock"
           Grid.Column="0" Grid.Row="4"/>
<TextBox Grid.Column="2" Grid.Row="4"/>
<StackPanel Grid.Column="0"
            Grid.ColumnSpan="3" Grid.Row="6"
            HorizontalAlignment="Right"
            Orientation="Horizontal">
    <Button x:Name="okButton" Width="75"/>
    <Button x:Name="cancelButton"
            Margin="5,0,0,0" Width="75"/>
</StackPanel>
<StackPanel Grid.Column="0" Grid.ColumnSpan="3"
            Grid.Row="8">
    <Button
        Click="ChangeLocalizationAndDoNotUpdate">
        Change localization and do not update UI
    </Button>
```

```
<Button Click="ChangeLocalizationAndUpdate"
        Margin="0,5,0,0">
    Change localization and update UI
</Button>
</StackPanel>
</Grid>
```

Результат приведенной выше разметки показан на рис. 11.

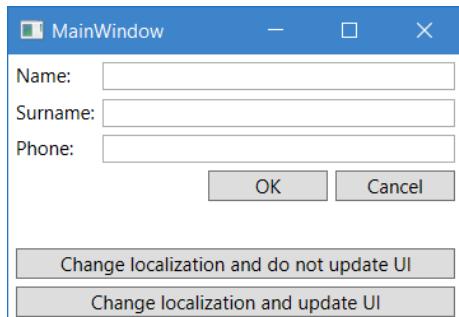


Рис. 11. Переключение локализации во время выполнения

Далее в соответствующем файле с программным кодом (модель code-behind) необходимо написать следующий фрагмент программного кода:

C#

```
private void ChangeLocalizationAndDoNotUpdate (
    object sender, RoutedEventArgs e)
{
    UpdateLocalization();
}

private void ChangeLocalizationAndUpdate(object
    sender, RoutedEventArgs e)
{
```

```
        UpdateLocalization();
        UpdateUI();
    }

private void UpdateLocalization()
{
    Thread.CurrentThread.CurrentCulture =
        new CultureInfo("es");
}

private void UpdateUI()
{
    nameTextBlock.Text = Strings.Name;
    surnameTextBlock.Text = Strings.Surname;
    phoneTextBlock.Text = Strings.Phone;
    okButton.Content = Strings.OK;
    cancelButton.Content = Strings.Cancel;
}

private void Window_Loaded(object sender,
                           RoutedEventArgs e)
{
    UpdateUI();
}
```

При создании новых вариантов локализаций не обязательно переопределять все виды ресурсов. Если при запуске приложения был выбран не основной файл ресурсов и там не оказалось требуемого ресурса, то будет автоматически использована его версия из главного файла ресурсов (файл, без суффикса с указанием локализации). Это может быть очень полезно когда локализация не очень сильно отличается. Например, основной версией языка может быть американский английский, а в локализованном файле с британским английским придется переопределить только отличающиеся строки, которых будет существенно меньше, чем одинаковых.

Приложение не ограничено одним набором файлов ресурсов (например, Strings.resx, Strings.fr.resx, Strings.es.resx). Часто строки разделяются на несколько независимых групп файлов ресурсов, чтобы их было легче поддерживать. Например, к данному проекту можно было бы добавить группу файлов, который содержали бы все возможные сообщения, выводимые пользователю во время работы приложения (Messages.resx, Messages.fr.resx, Messages.es.resx).

1.1.1. Терминология и сокращения

При внедрении локализации в программный продукт часто приходится описывать собственные классы, методы, пространства имен, создавать директории, хранящие локализуемые данные и т.д. При именовании перечисленных объектов, обычно, придерживаются определенных конвенций (правил), а именно, часто можно встретить следующие сокращения: **i18n** и **l10n**.

Если встретить их первый раз в чужом коде, то первым предположением, скорее всего, будет то, что программист выбрал произвольную комбинацию цифр и букв для имени типа данных или переменной, но это не так. Они являются сокращениями от двух слов: *internationalization* (интернационализация) и *localization* (локализация), соответственно. При этом сокращение формируется по особым правилам. Числа входящие в их состав означают количество букв располагающееся между первой и последней буквой слова. Например, **l10n**, это: первая буква слова (**l**), затем 10 букв (**ocalizatio**), и в конце последняя буква слова (**n**).

2. Расширения разметки

Описывая пользовательский интерфейс при помощи XAML-разметки разработчики часто сталкиваются с одной проблемой: правила разметки требуют указания всех значений в виде литералов, т.е. они должны быть статичными и известными на этапе верстки. Если, к примеру, требуется отобразить имя пользователя или его телефон в текстовом поле, то обойтись одной лишь разметкой здесь не получится, потому что имя пользователя и телефон представляют из себя информацию, которая меняется от пользователя к пользователю, т.е. они не статичны. Другая проблема заключается в том, что на этапе создания приложения еще нет никаких пользователей, т.е. их данные не известны вовсе. Решением данной проблемы был бы механизм, который позволял бы откладывать получение значения для атрибута на момент выполнения, например, определял бы аутентифицированного пользователя и использовал бы его данные для отображения в элементах управления.

На самом деле WPF предоставляет два механизма для решения описанной проблемы. Одним из них является использование специальных объектов-преобразователей, о которых речь пойдет в следующих разделах. Коротко говоря, эти объекты позволяют преобразовать строку в объект и, при необходимости, наоборот. Однако, существует множество сценариев, в которых необходимо совершенно другое поведение обработки предоставленного строкового литерала.

Например, может быть необходимость проинструктировать XAML-анализатор в том, чтобы он не создавал новый объект, обрабатывая разметку, а использовал один из ранее созданных, или же использовал статически созданный объект. Необходимость создания объекта используя параметризованный конструктор также встречается часто. Просто так в разметке нет возможности это указать, так как все описанные в ней объекты создаются используя конструктор типа данных по умолчанию, т.е. без параметров. Также бывают ситуации, при которых значение задаваемое свойству объекта, при помощи атрибута в XAML-разметке, не известно на этапе написания исходного кода, и должно быть вычислено на этапе компиляции. Все описанные ситуации и некоторые другие, не упомянутые здесь, разрешимы при помощи расширений разметки.

2.1. Создание собственных расширений разметки

Существует несколько стандартных расширений разметки, но перед их рассмотрением, для лучшего понимания происходящего процесса, стоит рассмотреть пример создания собственного расширения разметки.

Каждое расширение разметки описывается в виде отдельного класса, который должен быть производным от класса `System.Windows.Markup.MarkupExtension`. Данный класс является абстрактным и требует переопределение одного абстрактного метода `System.Windows.Markup.MarkupExtension.ProvideValue`. Когда XAML-анализатор встречает в разметке расширение разметки, то создается экземпляр соответствующего класса и вызывается вы-

шеупомянутый метод, результат которого присваивается свойству, которому соответствует обрабатываемый атрибут.

Собственные классы расширений разметки принято называть так, чтобы они заканчивались словом Extension, которое, при использовании в XAML-разметке, можно опускать.

Для того, чтобы указать в качестве значения атрибута расширение разметки, необходимо использовать особый синтаксис:

XAML

```
<Element Attribute="{MarkupExtensionName}" />
```

Здесь фигурные скобки указывают XAML-анализатору, что дальше идет применение расширения разметки, а MarkupExtensionName определяет, какое именно расширение разметки стоит использовать.

2.1.1. Пространства имен

Создавая собственные расширения разметки и во многих других случаях придется взаимодействовать с пространствами имен, в которых объявлены клиентские типы данных. Для подключения пространства имен в разметке необходимо использовать специальный атрибут в любом элементе. После описания пространства имен, оно станет доступно для использования в данном элементе и во всех вложенных, поэтому, чаще всего, пространства имен описываются в корневом элементе XAML-файла. Синтаксис описания пространства имен выглядит следующим образом:

XAML

```
<Element xmlns:XamlNamespace =
    "clr-namespace:CSharpNamespace;
    assembly=AssemblyName"/>
```

Данное описание состоит из трех частей. XamlNamespace задает идентификатор, который будет являться псевдонимом для существующего пространства имен из .NET Framework и будет использоваться в XAML-разметке. CSharpNamespace задает пространство имен из .NET Framework, в котором находится требуемый тип данных. AssemblyName указывает название сборки, в которой находится требуемое пространство имен. Если необходимо пространство имен находится в той же сборке, что и описание пространства имен в XAML-разметке, то часть с описанием сборки можно опустить:

XAML

```
<Element xmlns:XamlNamespace =
    "clr-namespace:CSharpNamespace"/>
```

После описания пространства имен, необходимо указывать его перед каждым упоминанием типа данных, находящегося в нем:

XAML

```
<Element xmlns:XamlNamespace =
    "clr-namespace:CSharpNamespace"/>
    <XamlNamespace:OtherElement/>
</Element>
```

2.1.2. Интерпретация текста при использовании расширений разметки

Строковые лексемы, располагающиеся после имени расширения разметки, но при этом все еще находящиеся в пределах фигурных скобок трактуются XAML-анализатором с учетом следующих правил:

- Запятая всегда является разделителем лексем.
- Если разделенные запятой лексемы не содержат символа оператора присваивания, то они расцениваются как аргументы, передаваемые в конструктор расширения разметки при создании экземпляра. При этом они должны соответствовать по типу данных и порядку расположения. XAML-анализатор определяет какой именно конструктор стоит вызвать по количеству указанных аргументов, а не по их типу, поэтому не следует объявлять перегруженные конструкторы с одинаковым количеством параметров для класса расширения разметки. В случае возникновения подобной ситуации поведение XAML-анализатора не определено, хотя на практике стоит ожидать выброс исключения.
- Если разделенные запятой лексемы содержат символ оператора присваивания, то они расцениваются как пары свойство-значение. В первую очередь вызывается конструктор по умолчанию для создания экземпляра расширения разметки, после чего происходит поочередное присваивание указанных значений указанным свойствам.
- Если встречается комбинация лексем содержащих и не содержащих символ оператора присваивания,

то описанные выше правила накладываются друг на друга, т.е. часть лексем может быть использована для указания аргументов конструктора, а часть для описания пар свойство-значение.

- Если возникает необходимость передать запятую в качестве части аргумента, необходимо заключить данный аргумент в одинарные кавычки.

2.1.3. Пример 1

В качестве примера будет рассмотрен процесс создания расширения разметки, которое будет возвращать название текущего дня недели в виде строки. Как было упомянуто ранее, необходимо описать класс, содержащий алгоритм получения требуемого значения. Ниже представлен программный код данного класса:

C#

```
internal sealed class TodayExtension : MarkupExtension
{
    public override object ProvideValue(
        IServiceProvider serviceProvider)
    {
        return DateTime.Now.DayOfWeek.ToString();
    }
}
```

После создания класса, описывающего расширение разметки, его можно использовать в любом месте XAML-разметки, при условии, что подключено пространство имен, в котором был описан класс.

Приведенный ниже фрагмент разметки демонстрирует использование созданного выше расширения разметки

(полный пример находится в папке Wpf.MarkupExtensions.Custom.Today):

XAML

```
<Window x:Class= "Wpf.MarkupExtensions.
    Custom.Today.MainWindow"
    ...
    xmlns:Local="clr-namespace:Wpf.
    MarkupExtensions.Custom.Today"
    ...
<TextBlock HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Run Text="Today: "/>
    <Run FontWeight="Bold" Text="{Local:Today}" />
</TextBlock>
</Window>
```

Результат приведенной выше разметки показан на рис. 12.

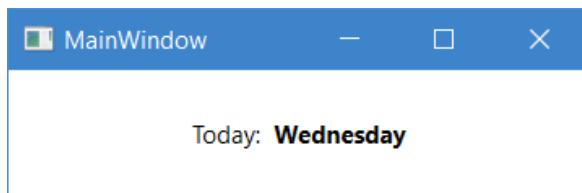


Рис. 12. Демонстрация клиентского расширения разметки

Анализируя XAML-разметку, во время обработки второго элемента `<Run>`, XAML-анализатор обнаружит, что для получения значения свойству `Text`, необходимо создать экземпляр расширения разметки `TodayExtension`, которое находится в пространстве имен `Local`, являющимся псевдонимом для пространства имен .NET Framework

Wpf.MarkupExtensions.Custom.Today. После создания будет вызван метод System.Windows.Markup.[MarkupExtension](#).ProvideValue, который вернет название текущего дня в виде строки.

2.1.4. Пример 2

Следующий пример будет возвращать название дня недели, который смещен на указанное количество дней от сегодняшнего. Также будет возможно настроить автоматический перевод результата в верхний регистр. Добиться такой функциональности возможно применив параметризованный конструктор и свойства для расширения разметки. Ниже представлен программный код данного класса:

C#

```
internal sealed class DayShiftExtension :  
    MarkupExtension  
{  
    private readonly int shift;  
  
    public DayShiftExtension() :  
        this(shift: 0)  
    {  
    }  
  
    public DayShiftExtension(int shift)  
    {  
        this.shift = shift;  
    }  
  
    public bool Uppercase { get; set; }  
  
    public override object ProvideValue(  
        IServiceProvider serviceProvider)  
    {
```

```
        string dayOfweek = DateTime.Now.  
            AddDays(shift).DayOfWeek.ToString();  
        if (UpperCase)  
        {  
            dayOfweek = dayOfweek.ToUpper();  
        }  
        return dayOfweek;  
    }  
}
```

Результат приведенной выше разметки показан на рис. 13.

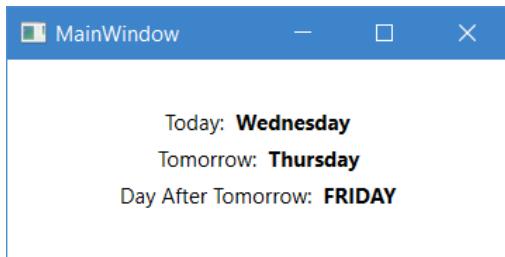


Рис. 13. Демонстрация клиентского расширения разметки

Приведенный ниже фрагмент разметки демонстрирует использование созданного выше расширения разметки (полный пример находится в папке Wpf.MarkupExtensions. Custom.DayShift):

XAML

```
<StackPanel HorizontalAlignment="Center"  
VerticalAlignment="Center">  
  
<TextBlock HorizontalAlignment="Center">  
    <Run Text="Today: "/>
```

```
<Run FontWeight="Bold"  
      Text="{Local:DayShift Uppercase=False}"/>  
</TextBlock>  
  
<TextBlock HorizontalAlignment="Center"  
           Margin="0,5,0,0">  
    <Run Text="Tomorrow: "/>  
    <Run FontWeight="Bold"  
          Text="{Local:DayShift 1}"/>  
</TextBlock>  
  
<TextBlock HorizontalAlignment="Center"  
           Margin="0,5,0,0">  
    <Run Text="Day After Tomorrow: "/>  
    <Run FontWeight="Bold"  
          Text="{Local:DayShift 2, Uppercase=True}"/>  
</TextBlock>  
  
</StackPanel>
```

2.2. Стандартные расширения разметки XAML

Существует несколько расширений разметки, которые не привязаны только к WPF. Ранее упоминалось, что XAML является самостоятельной технологией и может применяться и в схемах разработки отличных от WPF, например Silverlight, UWP (*Universal Windows Platform*) или Xamarin. Расширения разметки о которых речь пойдет далее принадлежат именно XAML, а не WPF. Они располагаются в сборке System.Xaml.dll и являются частью XAML-сервисов, реализованных в платформе .NET Framework. При этом они располагаются в пределах пространства имен XAML, и доступны в разметке через префикс x.

2.2.1. Расширение разметки x:Type

Расширение разметки x:Type (System.Windows.Markup.TypeExtension) принимает в качестве аргумента строку, содержащую название типа данных и возвращает соответствующий объект типа System.Type. Наиболее часто применяется при работе со стилями и шаблонами элементов управления. Данное расширение разметки играет роль, аналогичную оператору `typeof()` в C#.

Синтаксис использования в качестве значения атрибута:

XAML

```
<Element Property="{x:Type Prefix:TypeNameValue}">
```

Синтаксис использования в качестве элемента-свойства:

XAML

```
<Element>
    <Element.Property>
        <x:Type TypeName="Prefix:TypeNameValue"/>
    </Element.Property>
</Element>
```

В примерах синтаксиса, описанных выше, TypeNameValue является названием типа, для которого необходимо получить объект типа System.Type, а Prefix, является optionalным пространством имен, которое необходимо указывать при взаимодействие с типами данных, не доступными в пространстве имен, с которым работает XAML-анализатор.

Также допустим вариант синтаксиса с явным присваиванием значения свойству System.Windows.Markup.

TypeExtension.TypeName (хоть его использование и не является типичным при разработке):

XAML

```
<Element Property="{x:Type
    TypeName=Prefix:TypeNameValue}">
```

Пример использования расширения разметки:

XAML

```
<Style TargetType="{x:Type Button}">
    ...
</Style>

<Style>
    <Style.TargetType>
        <x:Type TypeName="Button"/>
    </Style.TargetType>
</Style>

<Style TargetType="{x:Type TypeName=Button}">
    ...
</Style>
```

2.2.2. Расширение разметки x:Null

Расширение разметки x:Null (System.Windows.Markup.NullExtension) возвращает **null** в качестве значения свойства. Используется, когда необходимо явным образом «убрать» значение у свойства ссылочного типа.

Синтаксис использования в качестве значения атрибута:

XAML

```
<Element Property="{x:Null}">
```

Пример синтаксиса использования в качестве элемента-свойства:

XAML

```
<Element>
    <Element.Property>
        <x:Null/>
    </Element.Property>
</Element>
```

Последний вариант синтаксиса применяется очень редко, так как данное расширение разметки не содержит опциональных параметров или параметризованных конструкторов.

Пример использования расширения разметки:

XAML

```
<Border Background="{x:Null}" />
<Border>
    <Border.Background>
        <x:Null/>
    </Border.Background>
</Border>
```

2.2.3. Расширение разметки x:Static

Расширение разметки x:Static (System.Windows.Markup.StaticExtension) возвращает значение статического члена типа. Используется, когда необходимо в качестве значения атрибута использовать значение, например, статического свойства другого класса.

Синтаксис использования в качестве значения атрибута:

XAML

```
<Element Property=
    "{x:Static Prefix:TypeName.StaticMemberName}">
```

Синтаксис использования как элемент-свойство:

XAML

```
<Element>
    <Element.Property>
        <x:Static Member= "Prefix:TypeName.
                            StaticMemberName"/>
    </Element.Property>
</Element>
```

В примерах синтаксиса, описанных выше, TypeName является названием типа, в котором находится требуемый статический член, а StaticMemberName выступает в роли названия этого статического члена. Prefix является optionalным пространством имен типа. При помощи данного расширения разметки можно обращаться к следующим членам типа:

- Константа.
- Статическое свойство.
- Статическое поле.
- Значение перечисления.

Также допустим вариант синтаксиса с явным присваиванием значения свойству System.Windows.Markup.StaticExtension.Member:

XAML

```
<Element Property = "{x:Static Member=
    Prefix:TypeName.StaticMemberName}">
```

Данное расширение разметки является очень удобным при локализации приложений, описанным выше способом. Однако, при этом есть очень важное ограничение. Расширение разметки `x:Static` может работать только с типами данных, объявленными со спецификатором доступа `public`. Класс, который генерируется для доступа к ресурсам по умолчанию обладает спецификатором доступа `internal`, который можно изменить на странице редактирования ресурсов (в верхней ее части).

Пример использования расширения разметки:

XAML

```
<TextBlock Text="{x:Static L10n:Strings.Greeting}">
<TextBlock>
    <TextBlock.Text>
        <x:Static MemberName="L10n:Strings.Greeting"/>
    </TextBlock.Text>
</TextBlock>
<TextBlock Text= "{x:Static Member=
    L10n:Strings.Greeting}"/>
```

2.2.4. Расширение разметки `x:Array`

Расширение разметки `x:Array` (`System.Windows.Markup.ArrayExtension`) предоставляет общий механизм для использования массивов объектов в пределах XAML-разметки.

Синтаксис использования в качестве элемента-свойства:

XAML

```
<Element>
    <Element.Property>
        <x:Array Type="TypeName">
```

```

    Contents
  </x:Array>
</Element.Property>
</Element>
```

Здесь TypeName указывает типом элементов в массиве, а Contents является содержимым, т.е. элементами массива, к которому можно добраться посредством свойства System.Windows.Markup.ArrayExtension.Items. В качестве значения свойства System.Windows.Markup.ArrayExtension.Type можно использовать расширение разметки x>Type, но также возможно применить сокращенный вариант, указав название типа в виде строки.

Пример использования расширения разметки:

XAML

```

<Window ...
  xmlns:System="clr-namespace:System;assembly=m
scorlib"
  ...>

  ...
  <x:Array Type="System:String">
    <System:String>Red</System:String>
    <System:String>Green</System:String>
    <System:String>Blue</System:String>
  </x:Array>

  <x:Array Type="{x>Type System:String}">
    <System:String>Red</System:String>
    <System:String>Green</System:String>
    <System:String>Blue</System:String>
  </x:Array>
  ...
</Window>
```

2.3. Стандартные расширения разметки WPF

В WPF существует несколько расширений разметки, которые привязаны к этой технологии и не являются частью XAML, и, следовательно, их не получится использовать в других технологиях, поддерживающих использование XAML. К наиболее часто используемым из них можно отнести расширения разметки для взаимодействия с ресурсами и те, которые помогают реализовать механизм привязки данных. В данном разделе будут рассмотрены некоторые из расширений разметки, применяемые для привязки данных. Остальные будут рассмотрены в будущих разделах.

2.4. Привязка данных

WPF обладает мощным механизмом привязки данных, который предоставляет простой и удобный способ для приложений представлять и обрабатывать данные. К элементу могут быть «привязаны» данные из разных источников посредством использования стандартных расширений разметки.

Привязка данных это процесс налаживания связи между пользовательских интерфейсом приложения и бизнес логикой (рис. 14). Если привязка настроена правильным образом и объекты-источники данных умеют генерировать необходимые оповещения при изменении данных, то обновления данных в пользовательском интерфейса будет происходить автоматически. Также привязка данных позволяет автоматически обновлять источник данных, если данные меняются через пользовательский интерфейс, к которому они были привязаны. Например,

при вводе текста в поле для ввода текста, можно получить автоматическое обновления данных в источнике.

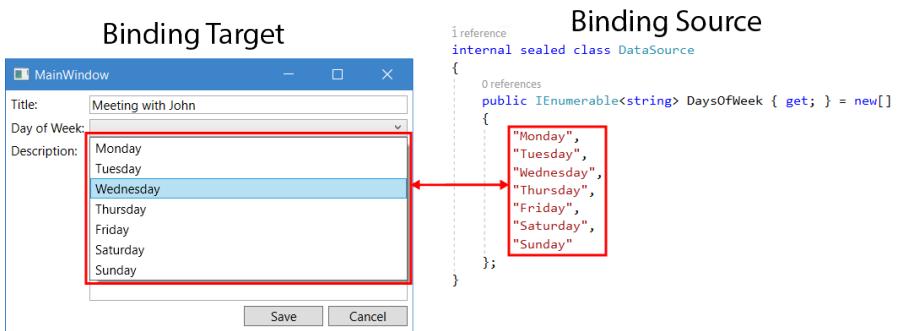


Рис. 14. Схема привязки данных

Несмотря на то, какой именно объект и источник данных используются в привязке данных, любая привязка данных всегда выглядит так, как это отображено на рис. 15.

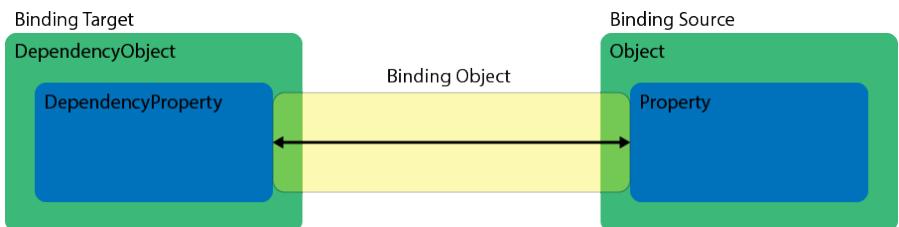


Рис. 15. Схема привязки данных.

Как видно из вышеприведенной схемы, привязка данных по существу является «мостом» между целевым объектом (*binding target*), к которому данные привязываются, и источником данных (*binding source*). Также отсюда можно вынести несколько ключевых понятий:

- Типично, каждая привязка данных состоит из четырех компонент: целевой объект привязки данных, целевое

свойство, источник привязки данных и путь к значению в источнике данных, которое необходимо использовать. Например, если представить, что необходимо привязать содержимое поля для ввода текста (TextBox) к свойству Name объекта класса Employee, то, целевым объектом будет TextBox, целевым свойством будет его свойство TextBox.Text, в качестве источника данных будет объект класса Employee, а требуемое значение будет содержать в его свойстве Employee.Name.

- Целевое свойство должно быть свойством зависимости (*dependency property*), речь о которых пойдет далее. Большинство свойств объектов типа System.Windows.UIElement и производных от него таковыми являются, и практически все из них, за исключением свойств доступных только для чтения (*read only*), поддерживают механизм привязки данных.
- Для того, чтобы реализовать привязку данных необходимо использовать промежуточный объект.

2.4.1. Направление потока данных

Как упоминалось выше, а также было показано на схеме привязки данных при помощи двунаправленной стрелки, поток данных может быть направлен в обе стороны. Например, данные в источнике могут обновляться после того, как пользователь вводит что-либо в поле для ввода текста. Также, при наличии специальных оповещений, текст в поле для ввода текста может измениться при изменении данных в источнике автоматически.

При создании объекта привязки данных есть возможность более тонко настроить поток данных, в зависимости

от ситуации, природы источника данных и возможностей целевого объекта (рис. 16).

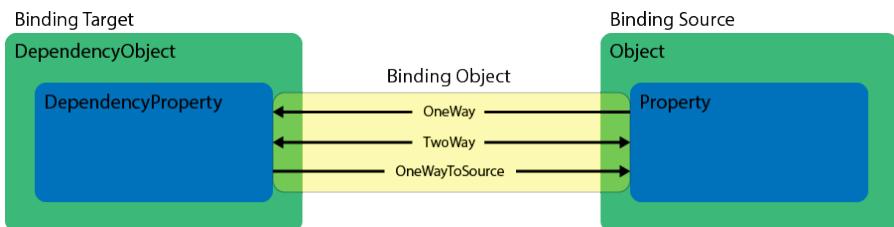


Рис. 16. Схема направлений потока данных

Для того, чтобы указать режим привязки данных, отвечающий за направление потока данных необходимо использовать перечисление `System.Windows.Data.BindingMode`, которое содержит следующие варианты:

- `Default`. Режим выбирается автоматически исходя из вида целевого объекта. В большинстве ситуаций выбирается режим `OneWay`, так как большая часть элементов управления предназначена только для отображения информации. Однако, для интерактивных свойств некоторых элементов управления (например, `System.Windows.Controls.TextBox.Text` или `System.Windows.Controls.CheckBox.IsChecked`) выбирается режим `TwoWay` для двунаправленного обновления данных.
- `OneTime`. Привязывает начальное значение из источника данных целевому объекту, но при этом не обновляется в будущем, если источник данных обновится и сгенерирует соответствующее оповещение. Является самым простым вариантом привязки, но при этом и самым «дешевым» в использовании.

Наиболее подходящим вариантом использования являются сценарии при которых данные в источнике не меняются.

- OneWay. Изменения в источнике данных автоматически отражаются в целевом объекте. Изменения в целевом объекте не передаются источнику данных. Данный вид привязки удобен в случае, когда целевой объект доступен только для чтения и, в принципе, не способен обновлять источник.
- OneWayToSource. Представляет из себя обратный вариант режима OneTime, т.е. при обновлении целевого объекта обновляется источник, но не наоборот.
- TwoWay. Изменения в источнике данных или в целевом объекте заставляют автоматически обновиться другой объект. Данный режим позволяет реализовать полностью интерактивный сценарии по работе с данными и автоматизировать их. Применяется в основном с элементами, предоставляющими возможность взаимодействия с данными пользователю: поля для ввода текста, использование переключателей или ползунков и т.д.

2.4.2. Расширение разметки Binding

Расширение разметки Binding (`System.Windows.Data.Binding`) применяется для создания привязки данных. Синтаксис использования в качестве значения атрибута:

XAML

```
<Element Property="{Binding PathString}">
```

Синтаксис использования в качестве элемента-свойства:

XAML

```
<Element>
    <Element.Property>
        <Binding Path="PathString"/>
    </Element.Property>
</Element>
```

В примерах синтаксиса, описанных выше, PathString представляет из себя строку, описывающую путь в объектном графе источника данных. Этот путь указывает к какому именно свойству в объекте-источнике данных необходимо обратиться для получения значения.

В качестве объекта-источника данных выступает объект, находящийся в свойстве System.Windows.FrameworkElement.DataContext. Данное свойство обладает типом System.Object и, следовательно, может содержать объект любого типа. Когда объекту привязки данных необходимо вычислить значение, которое необходимо установить целевому объекту происходит обращение к свойству System.Windows.FrameworkElement.DataContext объекта, для свойства которого применяется привязка данных. Если данное свойство не содержит никакого объекта (значение по умолчанию для данного свойства — **null**), то происходит обращение в аналогичное свойство родительского элемента в визуальном дереве и так до тех пор пока не будет найден объект у которого данное свойство будет заполнено каким-либо объектом. После этого произойдет обращение к свойству этого объекта согласно указанному пути.

Приведенный ниже фрагмент разметки демонстрирует использование расширения разметки для привязки данных (полный пример находится в папке Wpf. MarkupExtensions.Binding.NonNotifiableSource):

XAML

```
<Grid HorizontalAlignment="Center"
      VerticalAlignment="Center">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition Width="5"/>
        <ColumnDefinition Width="20"/>
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition/>
    </Grid.RowDefinitions>

    <TextBlock Grid.Column="0" Grid.Row="0"
              Text="Bound Value:"/>

    <TextBlock Grid.Column="2"
              Grid.Row="0"
              HorizontalAlignment="Center"
              Text="{Binding Value}"/>

    <TextBlock Grid.Column="0" Grid.Row="2"
              Text="Timer Value:"/>

    <TextBlock x:Name="textBlock"
              Grid.Column="2"
              Grid.Row="2"
              HorizontalAlignment="Center"/>
```

```

<Button Click="Button_Click"
        Grid.Column="0"
        Grid.ColumnSpan="3"
        Grid.Row="4">
    Start Timer
</Button>
</Grid>

```

Результат приведенной выше разметки показан на рис. 17.

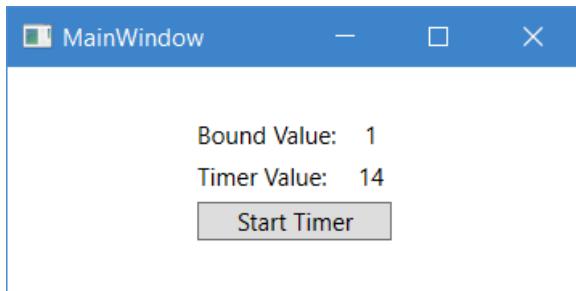


Рис. 17. Использование привязки данных без оповещений со стороны источника

Далее в соответствующем файле с программным кодом (модель code-behind) необходимо написать следующий фрагмент программного кода:

C#

```

internal sealed partial class MainWindow : Window
{
    private readonly DispatcherTimer timer;

    public MainWindow()
    {
        InitializeComponent();
    }
}

```

```
timer = new DispatcherTimer { Interval =
    TimeSpan.FromMilliseconds(300) };
timer.Tick += Timer_Tick;

DataContext = new DataSource();
}

private void Button_Click(object sender,
    RoutedEventArgs e)
{
    if (!timer.IsEnabled)
    {
        timer.Start();
    }
}

private void Timer_Tick(object sender,
    EventArgs e)
{
    ++((DataSource) DataContext).Value;

    UpdateTextBlock();
}

private void UpdateTextBlock()
{
    textBlock.Text = ((DataSource) DataContext).
        Value.ToString();
}

private void Window_Loaded(object sender,
    RoutedEventArgs e)
{
    UpdateTextBlock();
}
}
```

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа `DataSource`, объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource
{
    public int Value { get; set; } = 1;
}
```

Как видно из XAML-разметки и программного кода приложение работает следующим образом. В качестве текста для одного текстового поля используется привязка данных, а для второго начальное значение устанавливается при загрузке окна и далее обновляется при каждом срабатывании таймера. Таймер в свою очередь запускается при нажатии на кнопку.

Стоит подчеркнуть, что оба текстовых поля отображают значение одного и того же свойства `DataSource.Value`. При этом объект-источник создан в одном экземпляре, т.е. оба текстовых поля должны отображать одно и то же значение. Если же запустить приложение и нажать на кнопку для старта таймера, то можно будет заметить, что текстовое поле, которое обновляется посредством таймера отражает действительное значение свойства, а текстовое поле, использующее привязку данных корректно отразило лишь начальное значение. Все последующие обновления значения в источнике не отразились при помощи привязки данных.

Ранее упоминалось, что для корректной работы привязки данных объект-источник должен уметь «оповещать»

об изменении данных, чтобы соответствующие изменения отразились в интерфейсе. Так вот объект типа [DataSource](#) ничего подобного не делает при обновлении свойства [DataSource.Value](#). В данный момент может возникнуть вполне логичный вопрос: каким же образом стоит реализовать данное оповещение?

Для того, чтобы объект, использующийся для привязки данных мог понять, что объект-источник умеет оповещать об изменении данных необходимо, чтобы последний реализовывал интерфейс [System.ComponentModel.INotifyPropertyChanged](#). Данный интерфейс требует наличие всего лишь одного события [System.ComponentModel.INotifyPropertyChanged.PropertyChanged](#), которое должно генерироваться при изменении какого-либо из свойств объекта. При этом, вместе с событием передается название свойства, которое изменилось. Это дает возможность объекту-привязке подписаться на данное событие и ожидать изменения требуемого свойства. После чего считать обновленное значение из источника и отразить изменения в интерфейсе.

Ниже представлен модифицированный программный код объекта-источника (полный пример находится в папке [Wpf.MarkupExtensions.Binding.NotifiableSource](#)):

C#

```
internal sealed class DataSource :  
    INotifyPropertyChanged  
{  
    private int value = 1;  
    public int Value  
    {
```

```

        get => value;
        set
        {
            if (!this.value.Equals(value))
            {
                this.value = value;
                OnPropertyChanged(new
                    PropertyChangedEventArgs(
                        nameof(Value)));
            }
        }
    }

    public event PropertyChangedEventHandler
        PropertyChanged;

    private void OnPropertyChanged(
        PropertyChangedEventArgs e)
    {
        PropertyChanged?.Invoke(this, e);
    }
}

```

Показанная реализация интерфейса `System.ComponentModel.INotifyPropertyChanged` является достаточно полной и гибкой, хотя и не единственно правильной. При попытке реализовать данный интерфейс самостоятельно следует придерживаться определенных советов:

- Не стоит генерировать событие непосредственно в `set`-методе изменяемого свойства, а стоит вынести данное действие в отдельный метод, согласно рекомендаций Microsoft по реализации событий.
- Не стоит пренебрегать проверкой на равенство старого и нового значений свойства. Если убрать данное

условие и делать постоянное присваивание нового значение и генерирование события, не зависимо от того, является ли присваиваемое значение действительно новым, то можно получить существенное падение производительности и даже зависание пользовательского интерфейса. Дело в том, что каждый раз, когда генерируется событие `System.ComponentModel.INotifyPropertyChanged.PropertyChanged` выполняется сопутствующий программный код, для обновления интерфейса, которого может быть много. Поэтому генерировать данное событие необходимо только в том случае если значение действительно изменилось. Исключением может стать ситуация, где затраты на сравнение существенно больше, но такая ситуация встречается не часто.

- Придется отказаться от автоматических свойств C# при реализации интерфейса из-за необходимости внедрения программного кода в set-метод свойства. При этом объект-источник данных может содержать и свойства не участвующие в привязке данных. Такие свойства могут быть автоматическими при необходимости.
- Не стоит использовать в качестве аргумента для события название свойства в виде строкового литерала. Использование оператора `nameof()` из C# является в разы предпочтительнее, так как при допущении ошибки она будет выявлена еще на этапе компиляции. Также он отлично поддерживает возможный будущий рефакторинг идентификаторов (переименование).

При использовании привязки данных очень важно помнить, что свойства в объекте-источнике, на которые

идет привязка, обязательно должны быть публичными. Любой другой спецификатор доступа не сработает. Проблемой в данном случае является то, что такая программа (с некорректным спецификатором доступа) скомпилируется и даже отработает, но отработает некорректно. Дело в том, что весь механизм привязки работает через рефлексию и поиск свойств в источнике по именам. Если требуемое свойство не обнаруживается или оно является недоступным, то данная привязка просто игнорируется. Временами это очень затрудняет отладку, так как вся эта часть с автоматической привязкой скрыта в классах, которые являются частью WPF.

Если же используется один из видов привязки, который должен иметь возможность обновлять источник, и при этом, в источнике данных `set`-метод свойства отсутствует или недоступен, то приложение завершиться с ошибкой на этапе выполнения, в момент анализа разметки и попытки создания соответствующего объекта привязки.

2.4.3. Триггеры обновлений источника данных

Привязки данных использующие режимы `System.Windows.Data.BindingMode.TwoWay` или `System.Windows.Data.BindingMode.OneWayToSource` «слушают» изменения в целевом объекте и обновляют источник данных при необходимости. Например, элемент управления поле для ввода текста применяют двунаправленную привязку данных. Однако, когда именно происходит обновление источника данных: при вводе каждого нового символа или когда пользователь перейдет к работе с другим элементом, тем самым закончив вводить данные? За это

отвечает свойство System.Windows.Data.Binding.UpdateSourceTrigger.

Если значением данного свойство является System.Windows.Data.UpdateSourceTrigger.PropertyChanged, то источник обновляется сразу же после любого изменения данных. Если значением данного свойство является System.Windows.Data.UpdateSourceTrigger.LostFocus, то источник обновляется только после того, как элемент потеряет фокус ввода. Для большинства свойств значение настройкой по умолчанию является первый вариант, но обновление источника при нажатии на каждую клавишу для текстовых полей может быть слишком затратным. Поэтому по умолчанию они настроены на обновления источника после потери ими фокуса. Естественно, эту настройку можно изменить.

Приведенный ниже фрагмент разметки демонстрирует использование различных триггеров обновления источника данных (полный пример находится в папке Wpf.MarkupExtensions.Binding.UpdateSourceTrigger):

XAML

```
<Grid HorizontalAlignment="Center"
VerticalAlignment="Center">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="5"/>
        <ColumnDefinition/>
        <ColumnDefinition Width="5"/>
        <ColumnDefinition Width="100"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
```

```
<RowDefinition Height="5"/>
<RowDefinition/>
</Grid.RowDefinitions>
<TextBlock Grid.Column="0" Grid.Row="0"
           Text="Name:"/>
<TextBox Grid.Column="2"
           Grid.Row="0"
           Text="{Binding Name,
                  UpdateSourceTrigger=LostFocus}"
           Width="100"/>
<TextBlock Grid.Column="4" Grid.Row="0"
           Text="{Binding Name}"/>
<TextBlock Grid.Column="0"
           Grid.Row="2"
           Text="Surname:"/>
<TextBox Grid.Column="2"
           Grid.Row="2"
           Text="{Binding Surname, UpdateSourceTrigger =
                  PropertyChanged}"
           Width="100"/>
<TextBlock Grid.Column="4"
           Grid.Row="2"
           Text="{Binding Surname}"/>
</Grid>
```

Результат приведенной выше разметки показан на рис. 18.

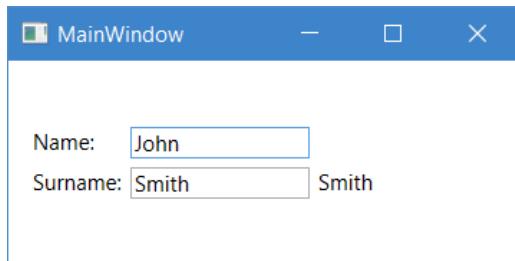


Рис. 18. Использование привязки данных без оповещений со стороны источника

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа **DataSource**, объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource :  
    INotifyPropertyChanged  
{  
    private string name = string.Empty;  
    private string surname = string.Empty;  
    public string Name  
    {  
        get => name;  
        set  
        {  
            if (!name.Equals(value))  
            {  
                name = value;  
                OnPropertyChanged(new  
                    PropertyChangedEventArgs(  
                        nameof(Name)))  
            }  
        }  
    }  
  
    public string Surname  
    {  
        get => surname;  
        set  
        {  
            if (!surname.Equals(value))  
            {  
                surname = value;  
                OnPropertyChanged(new  
                    PropertyChangedEventArgs(  
                        nameof(Surname)))  
            }  
        }  
    }  
}
```

```
        }
    }

public event PropertyChangedEventHandler
    PropertyChanged;

private void OnPropertyChanged(
    PropertyChangedEventArgs e)
{
    PropertyChanged?.Invoke(this, e);
}

}
```

2.4.4. Привязка коллекций

При работе со списковыми элементами управления механизм привязки данных обладает слегка другим набором требований, как со стороны XAML-разметки, так и со стороны источника данных.

При использовании привязки данных для того, чтобы указать набор объектов, который должен быть отображен в списковом элементу управления ([ListBox](#), [ComboBox](#), [TreeView](#) и др.) необходимо использовать свойство `System.Windows.Controls.ItemsControl.ItemsSource` вместо свойства `System.Windows.Controls.ItemsControl.Items`. При этом объект полученный из источника данных должен быть каким-то видом коллекции или массивом, т.е. должен реализовывать интерфейс `System.Collections.IEnumerable`. Однако, любая программная модификация такой коллекции не приведет к обновлению списка в пользовательском интерфейсе, даже если будет сгенерировано соответствующее

событие обновления свойства — System.ComponentModel.INotifyPropertyChanged.PropertyChanged.

Для того, чтобы получить синхронизацию с пользовательским интерфейсом, т.е. автоматическое его обновление при любой модификации привязанной коллекции, необходимо, чтобы данный объект-коллекция реализовывал интерфейс System.Collections.Specialized.INotifyCollectionChanged. Данный интерфейс содержит всего лишь одно событие, которое должно генерироваться при изменении коллекции. Именно на него будет совершена подписка во время создания объекта привязки данных. Хорошая новость заключается в том, что существует готовая коллекция, которая реализует данный интерфейс — System.Collections.ObjectModel.ObservableCollection.

Приведенный ниже фрагмент разметки демонстрирует использование различных видов коллекций при привязке данных к списковым элементам управления (полный пример находится в папке Wpf.MarkupExtensions.Binding.Collections):

XAML

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition Width="5"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="5"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
```

```

<ListBox Grid.Column="0" Grid.Row="0"
         ItemsSource="{Binding
                         NotifiableCollection}"/>
<ListBox Grid.Column="2" Grid.Row="0"
         ItemsSource="{Binding
                         NonNotifiableCollection}"/>
<Button Click="AddNotifiableValue_Click"
        Grid.Column="0"
        Grid.Row="2">
    Add Value with Notification
</Button>
<Button Click="AddNonNotifiableValue_Click"
        Grid.Column="2"
        Grid.Row="2">
    Add Value without Notification
</Button>
</Grid>

```

Результат приведенной выше разметки показан на рис. 19.

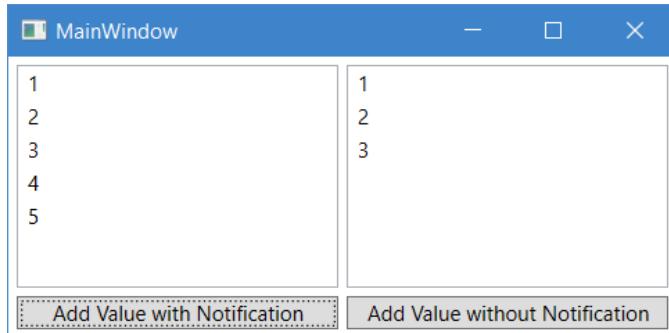


Рис. 19. Использование коллекция при привязке данных

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа

DataSource, объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource
{
    private readonly ICollection<int>
        nonNotifiableCollection =
        new List<int> { 1, 2, 3 };

    private readonly ICollection<int>
        notifiableCollection =
        new ObservableCollection<int> { 1, 2, 3 };

    public IEnumerable<int> NonNotifiableCollection =>
        nonNotifiableCollection;

    public IEnumerable<int> NotifiableCollection =>
        notifiableCollection;

    public void AddValueToNonNotifiableCollection()
    {
        nonNotifiableCollection.Add(
            nonNotifiableCollection.Count + 1);
    }

    public void AddValueToNotifiableCollection()
    {
        notifiableCollection.Add(
            notifiableCollection.Count + 1);
    }
}
```

3. Шаблоны данных

Как уже упоминалось ранее, для того, чтобы привязать коллекцию объектов к списковому элементу управления, необходимо использовать свойство `ItemsSource`. Однако, если остановиться лишь на этом, то будет применена модель отображения содержимого WPF, а так как в ситуации с привязкой данных, привязываются данные не производные от класса `System.Windows.UIElement`, то для них будет просто вызван метод `System.Object.ToString` и его результат поместиться в список.

Приведенный ниже фрагмент разметки демонстрирует использование спискового элемента управления, который не использует шаблон данных (полный пример находится в папке `Wpf.DataTemplates.WithoutDataTemplates`):

XAML

```
<ListBox ItemsSource="{Binding Employees}" Margin="5"/>
```

Результат приведенной выше разметки показан на рис. 20.

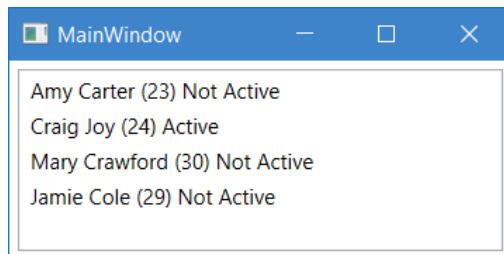


Рис. 20. Списковый элемент управления без шаблона данных

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа **DataSource**, объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource
{
    private readonly IEnumerable<Employee> employees;

    public DataSource()
    {
        employees = new []
        {
            new Employee("Amy", "Carter", 23),
            new Employee("Craig", "Joy", 24)
                { IsActive = true },
            new Employee("Mary", "Crawford", 30),
            new Employee("Jamie", "Cole", 29)
        };
    }

    public IEnumerable<Employee> Employees => employees;
}
```

Тип данных **Employee**, который используется в источнике данных выглядит следующим образом:

C#

```
internal sealed class Employee
{
    private readonly int age;
    private readonly string name;
    private readonly string surname;
```

```
public Employee(string name, string surname,
                int age)
{
    this.age = age;
    this.name = name;
    this.surname = surname;
}

public int Age => age;

public bool IsActive { get; set; }

public string Name => name;

public string Surname => surname;

public override string ToString()
{
    return $"{name} {surname} ({age})
{(IsActive ? "Active" :
"Not Active")}";
}
}
```

У такого решения есть несколько минусов. Во-первых, переопределять метод преобразования объекта в строку только для того, чтобы он корректно отображался в пользовательском интерфейсе не правильно, так как эта логика касается представления данных, а не самих данных. Во-вторых, если в разных местах необходимо разное отображение одного и того же объекта, этого уже не получится добиться, так как метод `System.Object.ToString` не может быть переопределен несколько раз для одного и того же типа данных. В-третьих, далеко не всегда подхо-

дит отображение объекта в виде строки. Довольно часто объект необходимо отобразить в виде набора элементов управления. Например, если это список пользователей, то для каждого пользователя в списке, возможно, необходимо отобразить его иконку, имя и кнопки «Редактировать» и «Удалить». В такой ситуации одной строкой не обойтись.

В WPF существует модель шаблонов данных, которая позволяет с большим уровнем гибкости описывать представление данных. При использовании привязки данных, это в общем-то основной способ получить представление данных отличное от обычного вызова метода `System.Object.ToString`.

Приведенный ниже фрагмент разметки демонстрирует использование спискового элемента управления, который использует шаблон данных (полный пример находится в папке `Wpf.DataTemplates.WithDataTemplates`):

XAML

```
<ListBox HorizontalContentAlignment="Stretch"
         ItemsSource="{Binding Employees}"
         Margin="5">

    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition/>
                    <ColumnDefinition Width="Auto"/>
                </Grid.ColumnDefinitions>
                <StackPanel Grid.Column="0"
                           Orientation="Horizontal">
                    <TextBlock>
```

```
<Run FontWeight="Bold"
      Text="{Binding Name,
      Mode=OneWay}"/>
<Run FontWeight="Bold"
      Text="{Binding Surname,
      Mode=OneWay}"/>
<Run Text="/">
    <Run Text="{Binding Age,
      Mode=OneWay}"/>
    <Run Text="")"/>
</TextBlock>
</StackPanel>
<CheckBox Grid.Column="1"
          IsChecked=
          "{Binding IsActive}">
    Is Active
</CheckBox>
</Grid>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
```

Результат приведенной выше разметки показан на рис. 21.

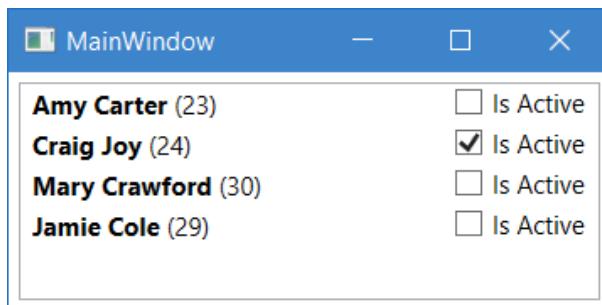


Рис. 21. Списковый элемент управления с шаблоном данных

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа **DataSource**, объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource
{
    private readonly IEnumerable<Employee> employees;

    public DataSource()
    {
        employees = new[]
        {
            new Employee("Amy", "Carter", 23),
            new Employee("Craig", "Joy", 24)
                { IsActive = true },
            new Employee("Mary", "Crawford", 30),
            new Employee("Jamie", "Cole", 29)
        };
    }

    public IEnumerable<Employee>
        Employees => employees;
}
```

Тип данных **Employee**, который используется в источнике данных выглядит следующим образом:

C#

```
internal sealed class Employee
{
    private readonly int age;
    private readonly string name;
    private readonly string surname;
```

```
public Employee(string name, string surname, int age)
{
    this.age = age;
    this.name = name;
    this.surname = surname;
}

public int Age => age;

public bool IsActive { get; set; }

public string Name => name;

public string Surname => surname;
}
```

Как видно из приведенного выше примера, для того, чтобы задать списковому элементу управления шаблон данных, необходимо описать его свойство `ItemTemplate`, в которое необходимо поместить объект типа `System.Windows.DataTemplate`, содержащий описание каждого элемента из списка. При создании элементов по шаблону, списковый элемент управления автоматически будет устанавливать каждый из объектов в качестве источника данных, поэтому применяя механизм привязки данных в пределах шаблона данных, необходимо привязываться к свойствам объектов, помещенных в коллекцию, привязанную к списковому элементу управления.

4. Команды

Приложения часто предоставляют несколько различных способ запуска одной и той же функциональности. В качестве примере можно представить такие известные команды, как «Копировать», «Вырезать» и «Вставить». Они присутствуют в любом приложение, поддерживающем обработку текстовых данных в том или ином виде. При этом активировать данные команды можно различными способами. Наиболее известный способ — использование горячих клавиш: `Ctrl+C`, `Ctrl+X` и `Ctrl+V`. Следующие способ — главное меню приложения (если таковое присутствует). Например, в Visual Studio это пункты меню: `Edit → Copy`, `Edit → Cut` и `Edit → Paste`. Также можно получить доступ к данным командам посредством контекстного меню элемента управления для ввода текста.

Во всех рассмотренных ситуациях, точки доступа к необходимым действиям совершенно разные: обработка клавиатуры, для обнаружения комбинаций горячих клавиш или обработка положения курсора и нажатия кнопок мыши для активации пункта меню. Однако, в результате необходимо получить одно и то же действие — команду.

WPF предоставляет несколько различных способов обрабатывать пользовательский ввод с различных устройств. В данном разделе пойдет о речь о командах, которые позволяют реализовать обработку пользовательского ввода на более абстрактном уровне, чем взаимодействие с устройствами ввода напрямую.

У команд есть несколько предназначений. Во-первых, это разделение объекта, который вызывает команду от логики, которая выполняет команду. Это позволяет некоторым совершенно различным объектам выполнять один и тот же алгоритм, а также позволяет настраивать этот алгоритм под каждый целевой объект по отдельности. Во-вторых, команды предоставляют интерфейс для того, чтобы можно было узнать доступна ли команда для выполнения. Если в качестве примере взять рассмотренные выше операции обработки текста, то операция «Вставить» будет доступна для выполнения только если в буфере обмена есть ранее скопированный текст, а операции «Копировать» и «Вырезать» ничего не будут выполнять при отсутствии выделения в тексте. В приложениях, обычно, для того, чтобы информировать пользователя о недоступности выполнения действия отключаются кнопки и пункты меню.

4.1. Создание команд

Все элементы WPF способные взаимодействовать с командами (кнопки, пункты меню и т.д.) требуют от клиентского программного кода реализации интерфейса `System.Windows.Input.ICommand`. Именно при помощи него и производятся все манипуляции с командами.

В WPF присутствует реализация данного интерфейса в виде класса `System.Windows.Input.RoutedCommand`, но в большинстве приложений ее будет недостаточно, поэтому в данном разделе будет рассмотрен вариант собственной реализации данного интерфейса с наиболее гибкой и расширяемой структурой.

Для начала стоит рассмотреть из чего состоит интерфейс команд, который предстоит реализовать.

Основным методом команды является метод System.Windows.Input.ICommand.Execute. Именно он вызывается при активации команды и содержит основной алгоритм команды (копирование, вырезание или вставка текста из примеров выше).

Также команда содержит метод System.Windows.Input.ICommand.CanExecute и событие System.Windows.Input.ICommand.CanExecuteChanged, которые работают сообща для реализации функциональности «отключения» команды. Метод вызывается во всех случаях, когда необходимо определить можно ли выполнять команду и возвращает значение типа System.Boolean, true если команду можно выполнять; иначе — false. Событие должно генерироваться, когда возможность выполнения команды изменяется. Имеется ввиду, что если команду, которую можно было выполнять по какой-то причине в текущий момент времени нельзя выполнить, то необходимо сгенерировать данное событие, чтобы заинтересованный программный код повторно «опросил» команду на предмет возможности выполнения и предпринял соответствующие действие (например, сделал неактивной кнопку регистрации или аутентификации).

4.1.1. Реализация интерфейса ICommand

Ниже приведена реализация интерфейса System.Windows.Input.ICommand.

C#

```
internal abstract class Command : ICommand
{
    public event EventHandler CanExecuteChanged;
```

```
protected virtual bool CanExecute()
{
    return true;
}

protected abstract void Execute();

protected virtual void OnCanExecuteChanged(
    EventArgs e)
{
    CanExecuteChanged?.Invoke(this, e);
}

public void RaiseCanExecuteChanged()
{
    OnCanExecuteChanged(EventArgs.Empty);
}

bool ICommand.CanExecute(object parameter)
{
    return CanExecute();
}

void ICommand.Execute(object parameter)
{
    Execute();
}
}

internal sealed class DelegateCommand : Command
{
    private static readonly Func<bool>
        defaultCanExecuteMethod = () => true;

    private readonly Func<bool> canExecuteMethod;

    private readonly Action executeMethod;
```

```
public DelegateCommand(Action executeMethod) :  
    this(executeMethod, defaultCanExecuteMethod)  
{  
}  
  
public DelegateCommand(Action executeMethod,  
    Func<bool> canExecuteMethod)  
{  
    this.canExecuteMethod = canExecuteMethod;  
    this.executeMethod = executeMethod;  
}  
  
protected override bool CanExecute()  
{  
    return canExecuteMethod();  
}  
  
protected override void Execute()  
{  
    executeMethod();  
}  
}
```

В приведенных выше двух классах заложена определенная функциональность и возможности к гибкому расширению.

Первый класс (**Command**) представляет базовую реализацию интерфейса. Как можно заметить основные методы интерфейса `System.Windows.Input.ICommand.Execute` и `System.Windows.Input.ICommand.CanExecute` представлены в виде явной реализации интерфейса. Это сделано для того, чтобы слегка изменить интерфейс команды. Дело в том, что эти методы в требуемом интерфейсе принимают параметр типа `System.Object`, который может использоваться в опре-

деленных ситуациях, которые в данном разделе не будут рассматриваться. Если его оставить в сигнатуре методов для будущих потомков, то это засорит их интерфейс. К тому же, данный параметр не строго типизирован, что заставляет клиентский код прибегать к явным преобразованиям типов данных, что в свою очередь очень часто приводит к ошибкам, так как проверки переносятся на этап выполнения, а не на этап компиляции. Для ситуаций требующих использование данного параметра, обычно, реализуется еще такая же пара классов-обобщений, которые используют данный параметр, но уже строго типизированный, тип которого указывается при создании команды.

Метод `Command.Execute` является абстрактным, так как нет никакой возможности определить ему алгоритм по умолчанию. Таким образом задача определения алгоритма полностью переносится на класс-потомок. Фактически, дочерний класс обязан реализовать лишь один этот метод, чтобы получить полностью рабочую команду с базовой функциональностью.

Метод `Command.CanExecute` по умолчанию возвращает `true`, так как очень часто встречаются команды, которые всегда могут выполняться, и, чтобы не обязывать производные классы описывать такую «заглушку», она описана в виде реализации метода по умолчанию. Если же команде требуется особы алгоритм для определения возможности своего исполнения, то необходимо лишь переопределить данный метод.

Также класс `Command` содержит всю необходимую инфраструктуру, необходимую для функционирования события `System.Windows.Input.ICommand.CanExecuteChanged`.

Второй класс ([DelegateCommand](#)) представляет собой особую версию команды, которая часто требуется. Дело в том, что распространена ситуация, когда алгоритм команды уже реализован в виде методов какого-либо класса и этот метод необходимо «обернуть» в интерфейс команды. Можно представить, что данный класс играет роль адаптера между методом класса и интерфейсом `System.Windows.Input.ICommand`.

Принципы здесь схожие с первым классом. Есть возможность внедрить два метода, которые будут исполняться при запросе активации команды и при проверке возможности ее активации. Также есть вариант упрощенного конструктора, который автоматически внедрит метод, всегда возвращающий `true`, для тех команд, который всегда активны.

Данная архитектура базовых классов команд дает две возможные точки расширения: производные классы от класса [Command](#), содержащие в себе алгоритм команды и методы, которые могут находиться в других классах и быть помещены внутрь объекта класса [DelegateCommand](#).

Приведенный ниже фрагмент разметки демонстрирует использование команд для кнопок и пунктов меню (полный пример находится в папке `Wpf.Commands.WithRaiseCanExecuteChanged`):

XAML

```
<DockPanel>
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="_Colors">
            <MenuItem Command="{Binding RedCommand}"
                      Header="_Red"/>
```

```
<MenuItem Command="{Binding GreenCommand}"  
          Header="_Green"/>  
<MenuItem Command="{Binding BlueCommand}"  
          Header="_Blue"/>  
</MenuItem>  
</Menu>  
<Grid Margin="5">  
    <Grid.ColumnDefinitions>  
        <ColumnDefinition/>  
        <ColumnDefinition Width="5"/>  
        <ColumnDefinition/>  
        <ColumnDefinition Width="5"/>  
        <ColumnDefinition/>  
    </Grid.ColumnDefinitions>  
    <Grid.RowDefinitions>  
        <RowDefinition/>  
        <RowDefinition Height="5"/>  
        <RowDefinition Height="Auto"/>  
    </Grid.RowDefinitions>  
    <Border Background="{Binding SelectedColor}"  
            Grid.Column="0"  
            Grid.ColumnSpan="5"  
            Grid.Row="0"/>  
    <Button Command="{Binding RedCommand}"  
           Grid.Column="0" Grid.Row="2">  
        Red  
    </Button>  
    <Button Command="{Binding GreenCommand}"  
           Grid.Column="2" Grid.Row="2">  
        Green  
    </Button>  
    <Button Command="{Binding BlueCommand}"  
           Grid.Column="4" Grid.Row="2">  
        Blue  
    </Button>  
</Grid>  
</DockPanel>
```

Результат приведенной выше разметки показан на рис. 22.

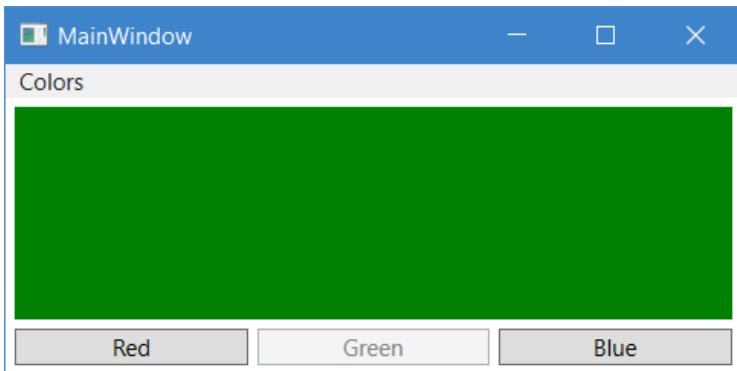


Рис. 22. Использование команд

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа `DataSource`, объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource :  
INotifyPropertyChanged  
{  
    private readonly Command blueCommand;  
    private readonly Command greenCommand;  
    private readonly Command redCommand;  
    private string selectedColor = "Black";  
  
    public DataSource()  
    {  
        blueCommand = new DelegateCommand(  
            () => SelectedColor = "Blue",  
            () => SelectedColor != "Blue"  
        );  
    }  
}
```

```
greenCommand = new DelegateCommand(
    () => SelectedColor = "Green",
    () => SelectedColor != "Green"
);

redCommand = new DelegateCommand(
    () => SelectedColor = "Red",
    () => SelectedColor != "Red"
);

PropertyChanged += (sender, e) =>
{
    if (e.PropertyName.
        Equals(nameof(SelectedColor)))
    {
        blueCommand.RaiseCanExecuteChanged();
        greenCommand.RaiseCanExecuteChanged();
        redCommand.RaiseCanExecuteChanged();
    }
};

public ICommand BlueCommand => blueCommand;

public ICommand GreenCommand => greenCommand;

public ICommand RedCommand => redCommand;

public string SelectedColor
{
    get => selectedColor;
    set
    {
        if (!selectedColor.Equals(value))
        {
            selectedColor = value;
        }
    }
}
```

```
        OnPropertyChanged(
            new PropertyChangedEventArgs(
                nameof(SelectedColor)));
    }
}

public event PropertyChangedEventHandler
    PropertyChanged;

private void OnPropertyChanged(
    PropertyChangedEventArgs e)
{
    PropertyChanged?.Invoke(this, e);
}
}
```

В приведенном выше примере происходит следующее. При разборе разметки XAML-анализатор обнаруживает, что пунктам меню и кнопкам устанавливаются команды. Для всех элементов, поддерживающих работу с командами этот процесс идентичен: при помощи привязки данных (*data binding*) необходимо привязать объект, реализующий интерфейс `System.Windows.Input.ICommand` к свойству с названием `Command`.

При первом отображении элемента управления, к которому привязана команда, сперва вызывается метод `System.Windows.Input.ICommand.CanExecute` от привязанной команды и определяется начальное состояние элемента управления. Если метод возвращает `false`, то элемент управления отключается. Таким образом нет необходимости явным образом использовать свойство `System.Windows.FrameworkElement.IsEnabled`. Также в этот

момент происходит автоматическая подписка на событие System.Windows.Input.ICommand.CanExecuteChanged привязанной команды для будущего обновления состояния элемента управления.

После того, как элемент управления будет задействован (например, произойдет нажатие кнопки) первым делом для привязанной команды определиться, допустимо ли ее выполнение при помощи все того же метода System.Windows.Input.ICommand.CanExecute. Если данный метод вернет `false`, то команда не будет выполнена; иначе — вызовется метод System.Windows.Input.ICommand.Execute.

В данном примере, при срабатывании любой команды происходит установка значения свойству `DataSource`.`SelectedColor`, что в свою очередь генерирует событие об изменении значения свойства, на которое подписан сам класс `DataSource`. Это сделано для того, чтобы после установки нового цвета сразу же выключить соответствующую команду. После срабатывания события изменения свойства `DataSource`.`SelectedColor` происходит вызов метода `Command.RaiseCanExecuteChanged`, который генерирует событие System.Windows.Input.ICommand.CanExecuteChanged.

На данное событие производилась автоматическая подписка со стороны WPF на этапе анализа XAML-разметки. Данный обработчик события, при срабатывании, снова вызывает метод System.Windows.Input.ICommand.CanExecute и, на основании его результата, выключает или выключает элемент управления.

Если не вызывать метод `Command.RaiseCanExecuteChanged` (полный пример находится в папке Wpf).

Commands.WithoutRaiseCanExecuteChanged), то кнопки не будут отключаться. Однако, при этом повторное нажатие на них не будет давать никакого эффекта, так как проверка возможности выполнения команды осуществляется всегда перед ее выполнением, независимо от состояния элемента управления, так как есть элементы, не способные отобразить свою неактивность, но при этом способные выполнять команды, например, горячие клавиши.

4.2. Горячие клавиши

Использование горячих клавиш для собственных приложений в WPF сводится к двум вещам: команды и привязка данных.

Для того, чтобы в примере рассмотренном выше переключать цвета можно было комбинациями горячих клавиш, например Ctrl+R, Ctrl+G и Ctrl+B, необходимо добавить следующий фрагмент XAML-разметки (полный пример находится в папке Wpf.Commands.HotKeys):

XAML

```
<Window.InputBindings>
    <KeyBinding Command="{Binding BlueCommand}"
        Key="B"
        Modifiers="Ctrl"/>
    <KeyBinding Command="{Binding GreenCommand}"
        Key="G"
        Modifiers="Ctrl"/>
    <KeyBinding Command="{Binding RedCommand}"
        Key="R"
        Modifiers="Ctrl"/>
</Window.InputBindings>
```

4.3. Команды для событий

WPF предоставляет отличную поддержку для команд в наиболее необходимых местах, таких как обработчик нажатия кнопки, активация пункта меню или использование горячих клавиш. Однако, не редко встречаются ситуации, когда необходимо выполнить какое-то действие (команду) при наступлении одного из события, описанного в элементах управления. Например, при первом запуске окна необходимо выполнить считывание объектов из файла и загрузить их в коллекцию, привязанную к пользовательскому интерфейсу. Из этого следует, что и коллекция и алгоритм загрузки не должны находиться в пределах XAML-разметки окна или в файле с программным кодом (модель code-behind), т.е. использование метода-обработчика события `System.Windows.FrameworkElement.Loaded` не представляется возможным. Для подобных ситуаций необходимо воспользоваться функциональностью преобразования события в команду.

Подобной встроенной функциональности в WPF нет, но Microsoft выпустила библиотеку под названием `System.Windows.Interactivity.dll`, которая ее реализует. Для того, чтобы она оказалась доступной в проекте можно воспользоваться NuGet, где она называется `System.Windows.Interactivity.WPF`. NuGet представляет из себя платформу через которую разработчики могут обмениваться полезным программным кодом в виде готовых к использованию библиотек.

Для того, чтобы добавить новую библиотеку используя NuGet, в первую очередь необходимо открыть менеджер

NuGet пакетов для проекта. Для этого необходимо выбрать пункт меню **Project → Manage NuGet Packages...** (рис. 23).

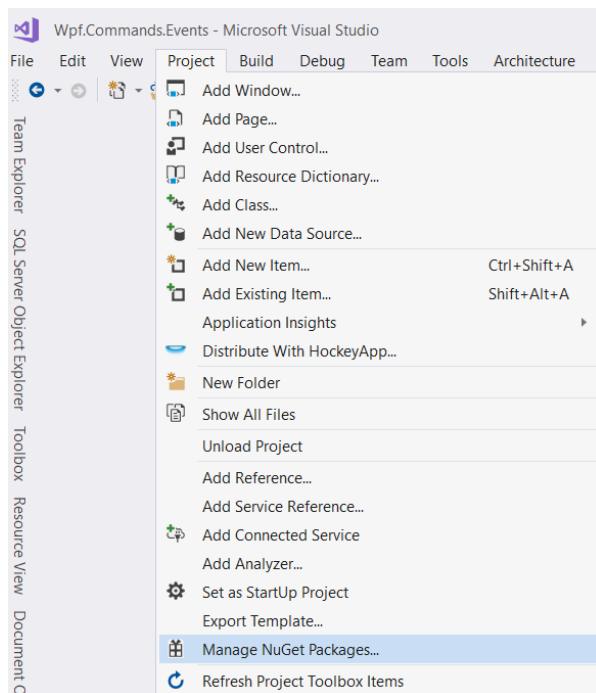


Рис. 23. Выбор пункта главного меню для добавления нового NuGet пакета

В открывшемся окне, слева сверху, необходимо выбрать **Browse**, после чего ввести имя требуемого пакета (`System.Windows.Interactivity` в данном случае) в поле для ввода текста, располагающееся сразу же под кнопкой **Browse**. После того, как загрузится список пакетов, которые удовлетворяют критерию поиска необходимо выбрать требуемый пакет (`System.Windows.Interactivity.WPF` в данном случае) и нажать кнопку **Install**, расположющуюся в правой части активного окна (рис. 24).

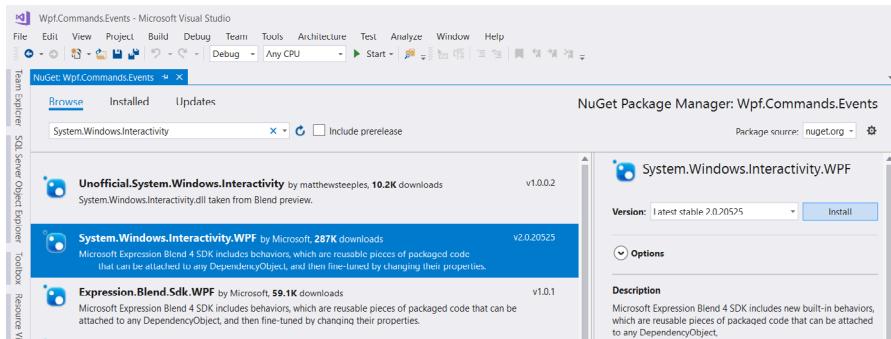


Рис. 24. Выбор NuGet пакета для добавления в проект

После того, как будет нажата кнопка **Install** начнется процесс установки, по завершению которого необходимая сборка (`System.Windows.Interactivity.dll` в данном случае) автоматически будет добавлена в проект (рис. 25).

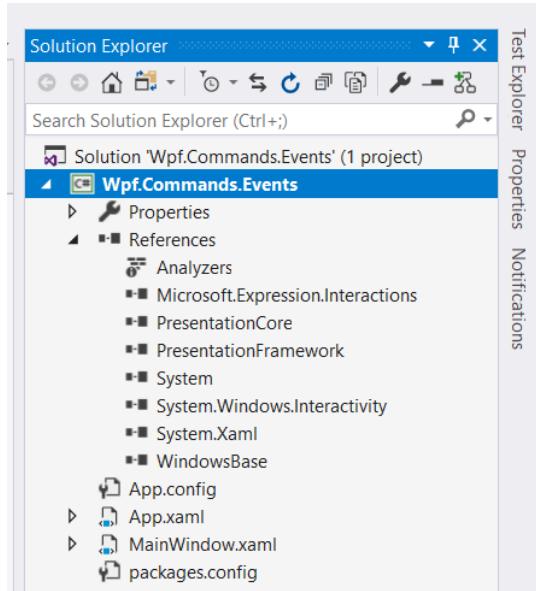


Рис. 25. Результат добавления NuGet пакета в проект

Приведенный ниже фрагмент разметки демонстрирует преобразование событий в команды (полный пример находится в папке Wpf.Commands.Events):

XAML

```
<Window ...>
    xmlns:Interactivity =
        "http://schemas.microsoft.com/
         expression/2010/interactivity"
    ...
    <Interactivity:Interaction.Triggers>
        <Interactivity:EventTrigger
            EventName="Loaded">
            <Interactivity:InvokeCommandAction
                Command="{Binding LoadedCommand}"/>
        </Interactivity:EventTrigger>
        <Interactivity:EventTrigger
            EventName="StateChanged">
            <Interactivity:InvokeCommandAction
                Command="{Binding
                    StateChangedCommand}"/>
        </Interactivity:EventTrigger>
    </Interactivity:Interaction.Triggers>

    <ListBox ItemsSource="{Binding Events}">
        <Interactivity:Interaction.Triggers>
            <Interactivity:EventTrigger
                EventName="SelectionChanged">
                <Interactivity:
                    InvokeCommandAction
                    Command= "{Binding
                        SelectionChangedCommand}"/>
            </Interactivity:EventTrigger>
        </Interactivity:Interaction.Triggers>
    </ListBox>
</Window>
```

Результат приведенной выше разметки показан на рис. 26.

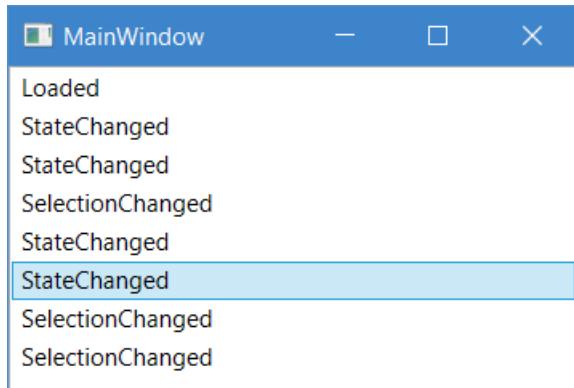


Рис. 26. Использование библиотеки System.Windows.Interactivity

Как должно быть заметно из примера разметки, для того, чтобы создать связку событие-команда, необходимо описать элемент EventTrigger с именем события для желаемого элемента.

В приведенном выше фрагменте программного кода, в качестве источника данных используется объект типа DataSource, объявление которого выглядит следующим образом:

C#

```
internal sealed class DataSource
{
    private readonly ICollection<string>
        events = new ObservableCollection<string>();
    private readonly ICommand loadedCommand;
    private readonly ICommand selectionChangedCommand;
    private readonly ICommand stateChangedCommand;
```

```
public DataSource()
{
    loadedCommand = new DelegateCommand(OnLoad);
    selectionChangedCommand =
        new DelegateCommand(OnSelectionChanged);
    stateChangedCommand =
        new DelegateCommand(OnStateChanged);
}

public IEnumerable<string> Events => events;

public ICommand LoadedCommand => loadedCommand;

public ICommand SelectionChangedCommand =>
    selectionChangedCommand;

public ICommand StateChangedCommand =>
    stateChangedCommand;

private void OnLoad()
{
    events.Add("Loaded");
}

private void OnSelectionChanged()
{
    events.Add("SelectionChanged");
}

private void OnStateChanged()
{
    events.Add("StateChanged");
}
```

5. Архитектурный шаблон проектирования MVVM

При создании приложений применяются разнообразные архитектурные шаблоны проектирования для упрощения разработки, поддержки и развития приложения в будущем. Идея разделения представления и бизнес логики приложения не нова в мире разработки программного обеспечения, ей уже более 30 лет. За последние годы архитектурные решения способствующие такому разделению становились все более и более популярны из-за нарастающей сложности программных продуктов и необходимости отображения пользовательских интерфейсов на широком спектре устройств, при этом повторно используя уже описанные бизнес правила и алгоритмы обработки данных.

5.1. Введение в архитектуру Model-View-X

Хорошо спроектированное приложение, это такое приложение, которое легко разрабатывать, тестировать, расширять и поддерживать. Для того, чтобы добиться перечисленных качеств программного продукта в нем должно быть разделение обязанностей и их инкапсуляция. Общим решением, обычно, является разделение пользовательского интерфейса (представления) от бизнес логики (модели). На самом деле, этот подход настолько распространен, что существует целый готовых ряд готовых решений, которые все вместе называются шаблонами проектирования Model-View (*Модель-Представление*).

5.1.1. Описание архитектурного шаблона проектирования MVC

Model-View-Controller (MVC). Архитектурный шаблон проектирования Model-View-Controller (*Модель-Представление-Контроллер*), который был представлен в 1970-х можно смело считать прародителем всего семейства. В данном шаблоне задачей Контроллера является создание Представлений и Моделей. Представление оповещает Контроллер о действиях пользователя (нажатия кнопок, ввод данных и т.д.), а Контроллер в свою очередь запрашивает состояние представление (например, что именно ввел пользователь) и обновляет соответствующим образом Модель (например, сохраняет введенные данные). Также Контроллер оповещает Представления в случае изменения Модели для того, чтобы они обновили свое состояние. Модель при этом представляет из себя данные и алгоритмы предметной области программного продукта. Именно модель отвечает за обработку и получение данных из конечного хранилища (базы данных, файла и т.д.).

Как было описано выше, архитектурный шаблон проектирования MVC состоит из трех компонентов: Model (*Модель*), View (*Представление*) и Controller (*Контроллер*). У каждого из них своя четко определенная роль. На рис. 27 показаны взаимосвязи данных компонентов.



Рис. 27. Устройство архитектурного шаблона проектирования MVC

5.1.2. Описание архитектурного шаблона проектирования MVVM

Model-View-ViewModel (MVVM). Архитектурный шаблон проектирования Model-View-ViewModel (*Модель-Представление-МодельПредставления*) является производным от MVC шаблоном проектирования, который активно задействует сильные стороны WPF для реализации промежуточного звена между Представлением и Моделью — Модели Представления. В MVVM Представление получает ссылку на Модель Представления, которая в свою очередь предоставляет через свой интерфейс набор команд и оповещаемых свойств, к которым представление «привязывается» используя механизм привязки данных рассмотренный ранее. Так как вся привязка данных и команд осуществляется для свойств объекта помещенного в `System.Windows.FrameworkElement`. `DataContext`, именно сюда записывается объект Модели Представления. Чтобы легче было понять и запомнить эту связь, стоит выучить следующее «уравнение MVVM»: `View.DataContext = ViewModel`. Далее, действия пользователя в Представлении генерируют срабатывание команд Модели Представления, а обновления Модели Представления передаются в Представление посредством оповещений и привязки данных.

Как было описано выше, архитектурный шаблон проектирования MVVM состоит из трех компонент: *Model* (*Модель*), *View* (*Представление*) и *ViewModel* (*МодельПредставления*). У каждого из них своя четко определенная роль. На рис. 28 показаны взаимосвязи данных компонентов.

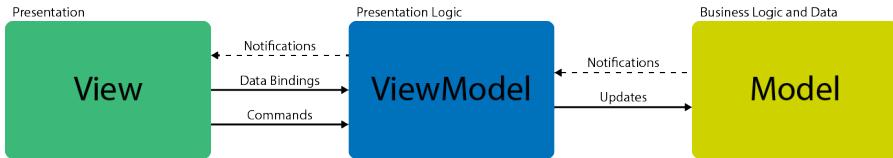


Рис. 28. Устройство архитектурного шаблона проектирования MVVM

Подробное разделение позволяет добиться следующего:

- Каждый компонент может быть легко подменен.
- Внутренняя реализация каждого компонента может быть изменена и это не потребует изменений в других компонентах.
- Каждый компонент может разрабатываться отдельно от остальных.
- Каждый компонент можно протестировать изолированно от остальных.

В дополнение к пониманию обязанностей каждого компонента также очень важно понимать принципы их взаимодействия. На самом высоком уровне находится Представление, которое «знает» о Модели Представления, а Модель Представления «знает» о Модели. При Модель не знает о Модели Представления, а Модель Представления не знает о Представлении.

Модель Представления изолирует Модель от Представления и позволяет первой развиваться независимо.

Представление

Представление ответственно за определение структуры и внешнего вида того, что пользователь будет видеть на экране (пользовательский интерфейс). В идеале, Пред-

ставление должно быть описано полностью на XAML с минимальным содержанием программного кода в соответствующем файле с программным кодом (модель code-behind). При этом данный программный код должен содержать только логику связанную с непосредственно этим Представлением (программное создание элементов управления, анимация и т.д.) и ни в коем случае не должен содержать бизнес логику приложения.

Представление может обладать собственной Моделью Представления или получить ее по наследству от родительского элемента. Представление получает данные посредством привязки к свойствам Модели Представления. Во время выполнения приложения Модель Представления может генерировать события, оповещающие об изменении своих свойств и Представление соответствующе на них реагирует — обновляет пользовательский интерфейс на основе новых данных, полученных через механизм привязки данных.

В случае возникновения необходимости в выполнении программного кода на стороне Модели Представление используются команды. Также команды применяются для обновления состояния интерфейса: включение или отключение связанных с ними элементов управления, в зависимости от того, можно ли выполнить команду в данный момент времени.

Модель

Модель представляет из себя реализацию предметной области программного продукта, которая включает данные, бизнес логику и логику валидации (проверка на

корректность). При проектирования этого слоя очень важно контролировать не попадание в него семантики Представления. Имеется в виду, что в сущностях Модели не должно быть никакой привязки к элементам управления пользовательского интерфейса, устройствам ввода, способе отображения данных и т.д.

Модель Представления

Модель представления играет роль посредника между Представлением и Моделью и отвечает за обработку логики Представления. Обычно, Модель Представления взаимодействует с Моделью посредством обычных вызов метод Модели. Далее Модель Представления поставляет полученные данные Представлению в том виде, в котором оно может их использовать, например, преобразовывает строки полученные от Модели в другой формат. Также Модель Представления отвечает за поставку реализации команд, которые будут вызываться при каких-либо взаимодействиях пользователя с Представлением. Например, нажатие кнопки может спровоцировать выполнение команды, хранящейся в Модели Представления. Дополнительно Модель Представления может быть ответственна за определение логического состояния приложения, которое будет отражено в Представлении, например, индикации того, что запущенная операция все еще выполняется.

Для того, чтобы Модель Представления могла корректно участвовать в схеме обмена данными с Представлением (одно- или двунаправленная привязка данных), она должна генерироваться событие `System.ComponentModel.INotifyPropertyChanged.PropertyChanged`. Для того, чтобы это условие было удовлетворено, Модель Представления

должна реализовывать интерфейс System.ComponentModel.INotifyPropertyChanged.

5.2. Обоснование использования архитектурного шаблона проектирования MVVM

Для того, чтобы обосновать необходимость применения архитектурного шаблона проектирования MVVM в WPF-приложениях, необходимо рассмотреть несколько примеров с типичными ошибками, и далее, рассмотреть, как они будут решены с помощью шаблона.

В качестве примера будет рассматриваться приложение для просмотра списка книг. Приложение будет обладать двумя Представлениями: окно, позволяющее просматривать список всех книг и окно, отображающее одну книгу из списка и обладающее возможностью передвигаться назад и вперед по списку книг (рис. 29).

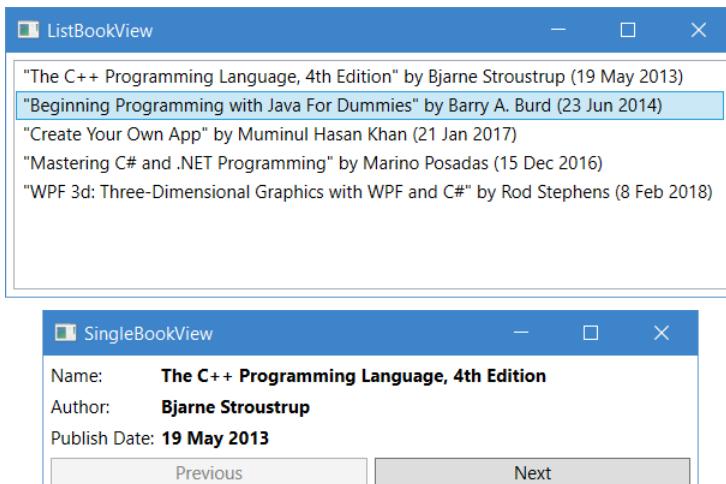


Рис. 29. Пользовательский интерфейс приложения для просмотра списка книг

Рассматривая программный код примеров не стоит обращать особое внимание на устройство и логику главного окна приложения. Оно необходимо, для запуска Представлений и возможности протестировать их поведение при добавлении новой книги. На главное окно не распространяется ни одно из предложенных ниже архитектурных решений. Далее приведен пример применения результирующего архитектурного решения в рамках всего приложения.

5.2.1. Упрощенная архитектура приложения

Для решения данной задачи разделение приложения на Модель и Представление кажется вполне естественным. Здесь список книг (библиотека) будет играть роль Модели, а два окна будут являться Представлениями.

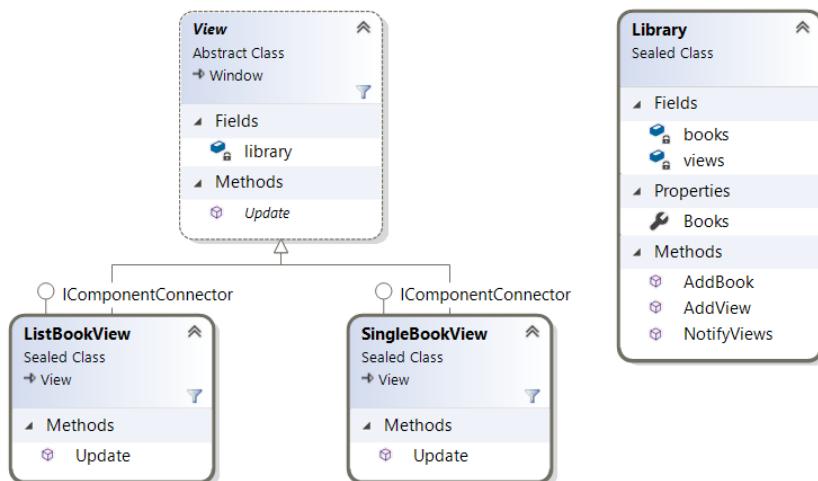


Рис. 30. Диаграмма классов приложения для просмотра списка книг

На рис. 30 показана диаграмма классов двухслойного решения Модель-Представление, в котором Представления

напрямую связаны с Моделью (полный пример находится в папке Wpf.Mvvm.Simplistic1).

Из приведенной выше диаграммы классов Модель содержит список Представлений, которые могут быть добавлены через соответствующий метод в интерфейсе (для простоты примера функциональность удаления опущена). Когда в Модель добавляется новая книга (или обновляется существующая), Модель оповещает все Представления, о которых она знает, вызывая метод View.Update, а Представления обновляют себя путем получения списка всех книг через свойство Library.Books. При создании экземпляров Представлений ListBookBook и SingleBookView им внедряется объект типа Library в качестве зависимости.

Стоит еще раз подчеркнуть, что Представление и Модель тесно связаны, а добавление новой бизнес логики еще больше их связывает.

5.2.2. Упрощенная архитектура приложения (WPF вариант)

Перед тем, как перейти к рассмотрению корректного решения, стоит преобразовать данный вариант приложения в «WPF-дружелюбный», путем использования команд и привязки данных.

На рис. 31 показана диаграмма классов обновленного решения, с использованием особенностей WPF (полный пример находится в папке Wpf.Mvvm.Simplistic2).

Теперь для Модели нет необходимости обновлять Представления, а следовательно, и знать о них. Вместо этого Представления содержат Модель в качестве источника данных (DataContext) и все обновления будут происходить автоматически из-за использованного механизма привязки данных.

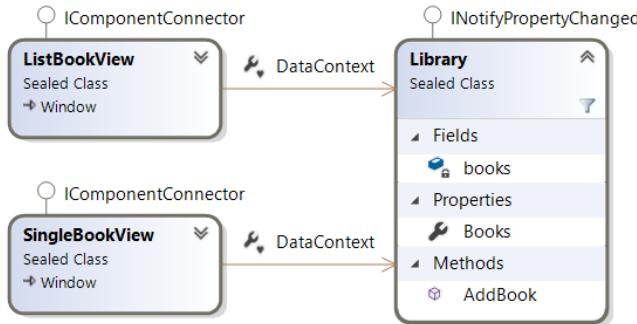


Рис. 31. Диаграмма классов приложения для просмотра списка книг

Из приведенной выше диаграммы классов должно быть заметно, что привязка данных позволяет получить более низкую связность компонентов.

5.2.3. Проблемы упрощенной архитектуры приложения

После обновления решения стало заметно, что часть логики была перенесена в модель, не смотря на то, что никакой новой функциональности с точки зрения предметной области приложение не получило. Теперь класс **Library** содержит не только список книг и методы управления этим списком, но также содержит еще и некоторую специфику Представления. Например, теперь ведется учет текущей выбранной книги и возможность переключению на следующую или предыдущую книгу. При этом данная функциональность необходима лишь одному из Представлений. Проблема в данном случае не в том, что другому Представлению подобная функциональность не надо, а в том, что весь этот программный код не относится к Модели, которая должна содержать только бизнес-логику программного продукта.

Также попытки добавления новой функциональности в одно из Представлений снова затронут модификацию Модели. Например, если потребуется реализовать возможность удаления книги из списка, то для этого потребуется добавить следующие недостающие компоненты в решение:

- Метод удаления книги из списка.
- Элемент управления для удаления (например, кнопка).
- Механизм определения выделенной книги в списке.
- Команда удаления, которая будет применяться для привязки алгоритма удаления к элементу управления.

Из данного списка, только первый пункт уместно помешать в Модель, так как это является прямым наращиванием функциональности Модели. Остальные пункты связаны с Представлением и его логикой. Элемент управления при этом, конечно же, будет помещен в необходимое место — Представление, и это будет правильным решением, но учет выделенной книги и команда удаления поместится в Модель, так как в данной архитектуре нет никакого другого подходящего для этого места, и это неправильно.

Данное архитектурное решение требует третий компонент, помимо Модели и Представления. Требуется «прослойка» между Моделью и Представлением, которая будет содержать логику работы Представления, при этом не вдаваясь в подробности того, как именно данные будут представлены пользователю. Имеется ввиду, что данному третьему участнику необходимо будет вести учет какая именно книга была выделена и какую команду необходимо использовать для ее удаления, но была она выбрана в элементе управления [ListBox](#), [ComboBox](#) или даже [TreeView](#) для него не должно иметь никакого значения.

5.2.4. MVVM архитектура приложения

Модель Представления (*View Model*) играет роль промежуточного слоя, предоставляя абстракцию, хранящую состояние, которое может использоваться несколькими Представлениями.

На рис. 32 диаграмма классов обновленного решения, с использованием Модели Представления в качестве промежуточного звена между Представлением и Моделью (полный пример находится в папке Wpf.Mvvm.Full).

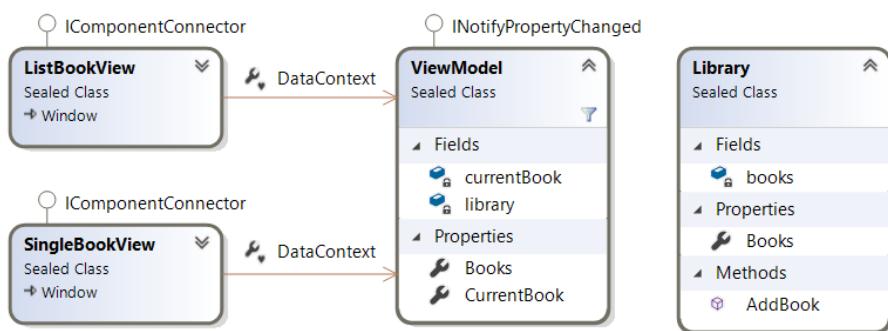


Рис. 32. Диаграмма классов приложения для просмотра списка книг

Используя такой подход Представление привязывается к свойствам и событию оповещения о их изменении Модели Представления. Модель Представления напрямую ничего не знает о Представлении, которое ее использует.

5.3. Достоинства применения архитектурного шаблона проектирования MVVM

MVVM позволяет достичь высокого качества совместного процесса разработки программистов и дизайнеров, и обладает следующими положительными качествами:

- Во время разработки программисты и дизайнеры могут работать более независимо и параллельно над своими компонентами.
- Программисты имеют возможность создать тесты для Модели Представления и протестировать всю логику без наличия Представления.
- Очень просто изменить пользовательский интерфейс приложения не затрагивая программный код, отвечающий за обработку логики, так как Представления практически полностью описываются при помощи XAML-разметки.
- При наличии готовой реализации бизнес логики (Модели), но при этом не полностью подходящей для использования в Представлении напрямую, нет необходимости менять Модель и случайно внести ошибки в до этого работающий программный код. Модель Представления служит своего рода адаптером между Моделью и Представлением, поэтому все необходимые преобразования или реализацию недостающих фрагментов могут быть помещены в нее.

5.4. Процесс внедрения MVVM в существующее приложение

В данном разделе будет рассмотрен процесс перевода приложения, в котором не применяется никаких архитектурных шаблонов проектирования на полноценное применение MVVM. Данный процесс будет произведен пошагово и каждый этап перевода будет сохранен в виде отдельного примера.

В качестве приложения будет использовано приложение, позволяющее вести учет своих контактов (рис. 33).

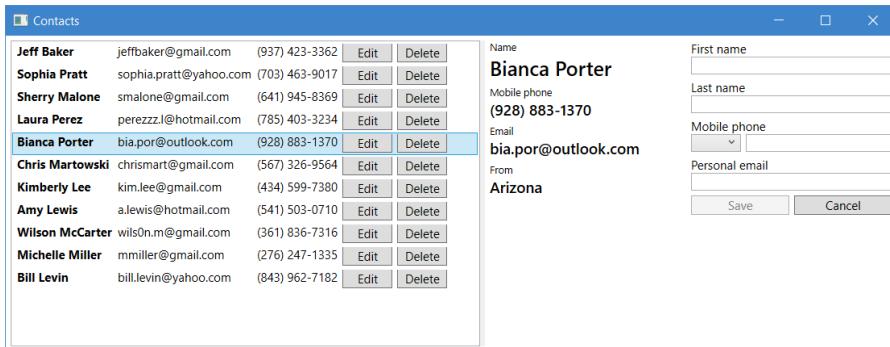


Рис. 33. Приложение для ведения учета контактов

Перед тем как перейти к рассмотрению программного кода, стоит обсудить особенности данного приложения, которые необходимо будет реализовать в каждом из примеров, независимо от применяемой архитектуры. Приложение обладает следующими особенностями:

- Все данные хранятся в XML-файле и сохранение происходит автоматически при любых изменениях со стороны пользовательского интерфейса (добавление, редактирование или удаление).
- Приложение поделено на три зоны: слева отображается список всех контактов, посередине отображается более детальная информация о выделенном контакте, а справа отображаются поля для ввода данных нового контакте или редактирования существующего.
- При редактировании контакты, совершенные изменения сразу же отображаются в пользовательском интерфейсе.

- Кнопка сохранения контакта становится активной только после того, как заполненные данные удовлетворяют минимальным критериям: имя, фамилия, почта и телефон должны быть заполнены, а также телефон должен состоять из семи символов.

Далее будет рассмотрено четыре варианта реализации данного приложения, каждый из которых будет строиться на основании предыдущего и улучшать его постепенно.

5.4.1. Вариант 1. Начальный

Первый вариант реализации данного приложения не использует никаких архитектурных шаблонов (полный пример находится в папке Wpf.Mvvm.Evolution.Initial). В данном примере моделью является класс [Contact](#).

Недостатки существующего программного кода:

- Представление тесно связано с Моделью.
- При добавлении новых Представлений, требующих уже существующие алгоритмы, их нельзя будет использовать повторно, так как они тесно связаны с текущим Представлением и не предполагают повторного использования.
- Тестирование логики Представления является крайне сложным если вовсе невозможным.
- Часть Представления описана в разметке, а часть в программном коде.
- Представление напрямую взаимодействует с источником данных в виде XML-файлом.
- Представление содержит описание пользовательского интерфейса, логику Представления и бизнес логику (логику Модели).

- Особенности WPF практически полностью игнорируются. Не используется привязка данных и команды.

5.4.2. Вариант 2. Задействование особенностей WPF

Второй вариант реализации данного приложения не использует никаких архитектурных шаблонов однако начинает более активно задействовать особенности WPF в виде привязки данных и команд (полный пример находится в папке `Wpf.Mvvm.Evolution.Step1`).

Изменения произведенные по сравнению с прошлым вариантом:

- Все обработчики событий преобразованы в команды и вынесены в отдельный класс.
- Обновление элементов пользовательского интерфейса производится посредством привязки данных.
- Недостатки существующего программного кода:
- Класс `DataSource` содержит логику Представления и логику Модели, что делает сложным их обновление по отдельности.
- Класс `Contact` содержит часть логики Представления. Это совершенно неуместно, а очень часто даже не возможно, так как класс может предоставляться третьей стороной и быть закрыт к изменениям или расширениям.

5.4.3. Вариант 3. Применение архитектурного шаблона MVVM

Третий вариант реализации данного приложения использует архитектурный шаблон MVVM (полный пример находится в папке `Wpf.Mvvm.Evolution.Step2`).

Изменения произведенные по сравнению с прошлым вариантом:

- Логика Представления и логика Модели четко разделена. Логика Представления полностью вынесена из класса **Contact**, который вернулся к варианту из первого примера.

5.4.4. Вариант 4. Выделение общего класса для Моделей Представления

Четвертый вариант реализации данного приложения использует архитектурный шаблон MVVM (полный пример находится в папке Wpf.Mvvm.Evolution.Final).

Изменения произведенные по сравнению с прошлым вариантом:

- Общая логика Моделей Представления вынесена в общий базовый класс.
- Введены атрибуты для более наглядного и простого указания зависимых свойств и методов.

6. Домашнее задание

Необходимо разработать приложение, которое позволяет настраивать цвет в системе ARGB и добавлять его в список цветов (рис. 34). Также у приложения есть возможность просмотреть список всех добавленных цветов и удалить некоторые при необходимости.

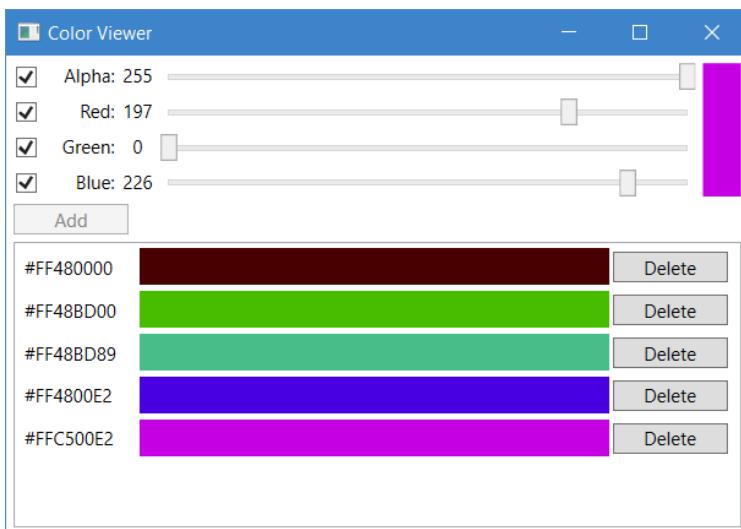


Рис. 34. Задание

Основным требованием при разработки данного приложения является применение архитектурного шаблона MVVM.

При запуске приложения пользователю отображается главное окно, на котором располагаются ползунки для выбора значений компонент цвета, кнопка добавления цвета и список сохраненных цветов. Также есть элемент,

визуализирующий выбранный цвет в реальном времени, т.е. при перетаскивании ползунка цвет обновляется.

Пользователь может сделать любой из ползунков неактивным, сняв отметку с элемента управления **CheckBox**, расположенного слева от него.

При нажатии на кнопку **Add** выбранный цвет добавляется в список сохраненных цветов, для отображения которых используется элемент **ListBox**.

Каждый сохраненный цвет может быть удален из списка цветов при помощи, соответствующей ему кнопки **Delete**.

При выборе пользователем цвета, который уже существует в списке цветов, кнопка **Add** должна становиться неактивной.



Урок № 4

Расширенные приемы работы с элементами управления

© Павел Дубский
© Компьютерная Академия «Шаг».
www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопротивления, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.