



**Разработка приложений
с использованием WPF**

Урок № 1

Введение в WPF.
Контейнеры.
Введение
в элементы
управления

Содержание

1. Описание WPF.....	5
2. Причины возникновения WPF	8
3. Архитектура WPF.....	17
4. Обзор WPF	21
4.1. XAML	21
4.2. Деревья пользовательского интерфейса	25
4.3. События и команды	28
4.4. Элементы управления	31
4.5. Примитивные элементы.....	32
4.6. Панели.....	34
4.7. Документы нефиксированного формата.....	35
5. Обзор XAML	37
5.1. Синтаксис XAML	37
5.2. Модель Code-Behind.....	59
6. Обзор базовых классов элементов визуального дерева	62

6.1. Класс DispatcherObject	62
6.2. Класс DependencyObject	64
6.3. Класс Visual.....	64
6.4. Класс UIElement.....	64
6.5. Класс FrameworkElement	65
6.6. Класс Panel	65
6.7. Класс Control	65
6.8. Класс ContentControl.....	65
6.9. Класс ItemsControl.....	66
7. Обзор WPF-приложения.....	67
7.1. Создание WPF-проекта	67
7.2. Структура WPF-приложения.....	68
8. Обзор редактора XAML.....	85
9. Компоновка.....	91
9.1. Этапы компоновки	93
10.Аппаратно-независимые единицы	97
11.Панели	98
11.1. Панель StackPanel	99
11.2. Панель WrapPanel.....	104
11.3. Панель DockPanel.....	108
11.4. Панель Grid.....	112
11.5. Панель UniformGrid	129
11.6. Панель Canvas	131
12.Позиционирование элементов	135
12.1. Отступы.....	137
12.2. Выравнивания	141
12.3. Размеры.....	148
13.Кнопки.....	157

13.1. Элемент управления Button	160
13.2. Элемент управления RepeatButton	161
13.3. Элемент управления CheckBox	163
13.4. Элемент управления RadioButton	166
14. Поля для ввода текста.....	171
14.1. Элемент управления TextBox	178
14.2. Элемент управления PasswordBox	182
15. Домашнее задание.....	186
Задание 1.....	186
Задание 2.....	186

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

1. Описание WPF

Windows Presentation Foundation (или WPF) — это программная платформа, разработанная компанией Microsoft для создания приложений, обладающих графическим пользовательским интерфейсом (GUI-приложения), которая принесла ряд фундаментальных изменений в процесс работы этих приложений, по сравнению с ее предшественниками.

На протяжении многих лет GUI-приложения, создаваемые для операционных систем семейства Windows, для своей работы использовали библиотеки USER32 и GDI32, которые предоставляют разнообразные оконные и графические сервисы, и располагаются в [user32.dll](#) и [gdi32.dll](#) соответственно. Эти библиотеки являются прямыми потомками своих соответствующих 16-битовых версий (USER16 и GDI16), которые были созданы в середине 1980-х годов. Разнообразные платформы для разработки пользовательских интерфейсов, созданные до выпуска WPF, (например, Windows Forms) являлись лишь «обертками» над этими библиотеками. WPF отличается от них тем, что совершенно не использует библиотеку GDI32 для отображения графических элементов пользовательского интерфейса и использует минимум функциональности из библиотеки USER32, тем самым, представляя из себя нечто большее, чем просто «обертку» над старым API.

WPF является частью .NET Framework и включает в себя фрагменты, как управляемого, так и неуправляемого кода. Несмотря на это, платформа WPF спроектирована

для использования только в контексте управляемой среды .NET Framework и не содержит открытого API для взаимодействия из неуправляемого кода. Следовательно, если вы хотите использовать WPF, вы обязаны использовать .NET Framework. Из-за того, что WPF является «родным» API для среды .NET Framework, получается избежать ряд неудобств, связанных с использованием старого, неуправляемого API, который выглядит неестественно в этой среде.

Начальная версия WPF была выпущена как часть платформы .NET Framework 3.0 в 2006 году. Так как WPF является частью платформы .NET Framework, их версии совпадают, т.е. если речь идет о WPF 3.5, имеется в виду версия WPF выпущенная вместе с .NET Framework 3.5. По этой причине начальную версию называют WPF 3.0.

Операционная система Windows Vista содержит предустановленную версию .NET Framework 3.0. Необходимо будет установить .NET Framework соответствующей версии на пользовательскую машину для того, чтобы запускать WPF-приложения на более ранних версиях Windows или для того, чтобы использовать более новую версию WPF.

Во всех случаях работает одно и то же правило: для того, чтобы WPF-приложение могло запуститься на целевой машине, на ней должна быть установлена платформа .NET Framework хотя бы той же версии, что была выбрана как целевая при создании приложения. Ниже представлен список версий .NET Framework и список операционных систем Windows, где они являются предустановленными.

.NET Framework	Windows	Windows Server	Visual Studio
.NET Framework 3.0	Vista	2008 SP2, 2008 R2 SP-1	—
.NET Framework 3.5	7, 8*, 8.1*, 10*	2008 R2 SP1	2008
.NET Framework 4.0	—	—	2010
.NET Framework 4.5	8	2012	2012
.NET Framework 4.5.1	8.1	2012 R2	2013
.NET Framework 4.5.2	—	—	—
.NET Framework 4.6	10	—	2015
.NET Framework 4.6.1	10 v1511	—	2015 Update 1
.NET Framework 4.6.2	10 v1607	2016	—
.NET Framework 4.7	10 v1703	—	2017

* Платформа .NET Framework 3.5 не является предустановленной на версиях операционной системы Windows 8, 8.1 и 10. Однако, может быть установлена через панель управления или любым другим способом.

2. Причины возникновения WPF

Любая новая технология или платформа в мире разработки программного обеспечения создается как средство решения какой-то проблемы. WPF является платформой разработки GUI-приложений и, следовательно, решает проблемы, связанные именно со спецификой разработки таких приложений. Библиотеки USER32 и GDI32, о которых шла речь ранее, и их API хорошо себя зарекомендовали за время своего существования и использования во множестве приложений, а также являлись широко распространенными и хорошо известными среди разработчиков. Зачем же в таком случае создавать еще одну подобную платформу, да еще и отказываться от проверенных средств разработки?

Дело в том, что слишком много всего изменилось с того времени как концепции, стоящие за старым пользовательским интерфейсом, были заложены в основу API. Компания Microsoft тратит очень много усилий для того, чтобы каждая новая версия Windows обладала как можно большей совместимостью со своими предшественниками. В качестве примера можно привести переход с 16-битной на 32-битную операционную систему Windows в 1990-х годах, одной из целей которого было сделать 32-битный API как можно более похожим на 16-битный, чтобы уменьшить проблемы портирования приложений под новую версию операционной системы.

В теории, можно было написать один раз исходный код, который мог бы успешно компилироваться как под 16-битную, так и под 32-битную версию Windows. Те программисты, которым удалось «насладиться» всеми радостями такого портирования могли бы возразить тому, на сколько легко это было на практике, но, тем не менее, это технически было возможно. Это означает, что библиотеки USER32 и GDI32 уходят своими корнями в далекие первые 16-битные версии Windows, которые были выпущены в середине 1980-х годов. Из этого следует, что решениям, принимавшимся при их проектировании, уже более четверти века.

Аппаратные средства прошли долгий путь с того времени, в частности видеокарты полностью изменились по своей природе. Во времена создания первой версии операционной системы Windows видеокарта представляла из себя довольно простое устройство, содержащее оперативную память в размере не большем, чем необходимый для запоминания информации, которую необходимо показать на экране, плюс немного электроники, для вывода этой информации на экран.

На сегодняшний день видеокарты обладают существенно большей мощностью. Размер оперативной памяти на них превышает размер всей доступной памяти машины времен первой Windows. Также процессорная мощность видеокарт во много раз больше чем 25 лет назад. Первые видеокарты работали на руку Windows, так как они создавались специально для того, чтобы операционные системы Windows, OS/2 и X Window System работали быстрее. Однако, со временем, производители видеокарт

поняли, что, сосредоточившись на 3D-ускорителях можно получать большую прибыль.

К сожалению, вся мощь новых видеокарт оказалась недоступной для разработчиков приложений, если только они не были готовы писать программный код с использованием OpenGL или DirectX. Несмотря на то, что многие так и делают, использование данного API добавляет слишком много проблем связанных со спецификой своего использования и не очень хорошо подходит для создания привычных элементов управления пользовательского интерфейса, так как он, по большей части, ориентирован на работу с 3D-графикой, в то время как интерфейсы большинства приложений используют 2D-графику для своего отображения. Архитектура библиотек USER32 и GDI32 оказалась не подходящей для использования новых возможностей 3D-ускорителей и поэтому видеокарты по большей части пристаивают во время работы приложений, использующих для вывода информации GDI или GDI+.

Одной из целей WPF является задействование этих неиспользуемых ресурсов при работе с привычными пользователю интерфейсами и элементами управления. Так как WPF в момент своего создания не был «скован» принципами которых придерживались более старые технологии, намного больше возможностей современных видеокарт получилось задействовать.

Видеокарты не единственное, что изменилось за время существования GUI-приложений. Для отображения первых интерфейсных приложений использовались мониторы на основе электронно-лучевой трубки (ЭЛТ-мониторы),

которые на сегодняшний день практически невозможно встретить, так как они были вытеснены жидкокристаллическими (ЖК-мониторы) и другими «плоскими» мониторами. Интересной особенностью технологии, применяемой при создании таких мониторов, является возможность достижения гораздо более высокого разрешения. ЭЛТ мониторы обладали максимальной плотностью пикселей (DPI) около 100 пикселей на дюйм, в то время как современные мониторы уже перешли черту в 200 пикселей на дюйм, и это не является окончательной границей. При такой высокой плотности пикселей человеческий глаз перестает различать отдельные пиксели, что делает изображение куда более приятным.

Основной причиной (кроме цены) почему такие мониторы не заполонили весь мир после своего появления является то, что приложения, написанные с использованием технологий-предшественников, не очень хорошо работают с пикселями такого размера, так как все они полагают, что пиксель всегда одного и того же размера на любом мониторе. Если подключить монитор с большей плотностью пикселей, каждый пиксель будет меньше, что сделает пользовательский интерфейс приложения также меньше. Это и является основной проблемой подобных мониторов. Интерфейс приложения становится настолько мелким, что само приложение становится практически полностью непригодным для использования.

По правде говоря, операционные системы Windows уже довольно давно предоставляют возможность программным путем узнать плотность пикселей используемого монитора, что в свою очередь делает возможным

написание программ, которые могли бы самостоятельно масштабировать собственный интерфейс в зависимости от используемого устройства. Проблема в том, что никто этим не заморачивается. Исторически сложилось, что все мониторы использовали пиксели практически одинакового размера и, следовательно, зачем кому-либо надо было тратить время и деньги на разработку программного обеспечения, которое могло бы поддерживать отображение на мониторах, которых не существовало. WPF-приложения призваны прервать этот цикл, являясь изначально автоматически масштабируемыми (*DPI-aware*). В отличие от разработчиков, которые применяют более старые технологии разработки, WPF-разработчик не должен предпринимать никаких специальных шагов, чтобы сделать свое приложение автоматически масштабируемым. На самом деле, придется предпринимать особые меры, что WPF-приложение перестало автоматически адаптироваться под разные размеры пикселей.

За более чем 25 лет изменились не только аппаратные средства, но также изменились требования пользователей к приложениям. В 1980-х годах для приложений было вполне приемлемо обладать интерфейсом командной строки (CLI-приложения). Более того, многие люди думали, что вся эта идея с «окнами» легкомысленная и акцентируется больше на внешнем виде, чем на содержимом. Однако, на сегодняшний день интерфейсы командной строки скорее считаются реликвиями прошлого, чем основным направлением проектирования интерфейсов приложений. Они по-прежнему популярны у определенных групп людей, к которым, в основном, относятся пользователи UNIX

платформ и любители использования командных строк, но для обычных пользователей такие интерфейсы больше не приемлемы. Они ожидают либо веб-интерфейс, либо оконный интерфейс.

Другой причиной выросших требований к интерфейсам стала сфера развлечений. В кино часто применяются интерфейсы несуществующих в реальности операционных систем и приложений, которые, тем не менее, выглядят в разы привлекательней невзрачных, стандартных, которые присущи более ранним версиям Windows. В качестве другого примера можно привести видеоигры, которые также обладают элементами пользовательского интерфейса, которые, кстати, в отличии от кино, полностью функционируют, и также выглядят красивее, чем в обычных приложениях.

Другим толчком в сторону повышения требований к графической составляющей приложений является развитие веб-индустрии. Веб-сайт практически любой компании выглядит в стиле присущем данной компании, что требует использования дизайна согласующегося с ее рекламой, буклетами и прочим, посредством чего компания представляет себя. Те веб-сайты, которые не придерживаются данной политики, выглядят любительскими на сегодняшний день. Обычные Windows приложения по сравнению с этим выглядят довольно невзрачно. Дело в том, что добиться подобного эффекта в плане графики и дизайна в приложениях такого рода намного сложнее, чем в веб-среде. И во многом веб обязан этому HTML и сопутствующим технологиям, которые позволяют даже людям, не обладающим навыками программирования создавать достаточно

сложные и привлекательные интерфейсы. При этом они даже не обязаны знать HTML. Существует множество инструментов для «визуальной» разработки сайтов, которые сами создают необходимую HTML-разметку.

Когда же речь заходит о том, что для разработки приложения под Windows необходимо найти человека, который понимает принципы работы оконного приложения и в состоянии сделать так, чтобы оно еще и выглядело привлекательно, шансы найти такого человека стремительно уменьшаются. WPF решает эту проблему путем схожим на решение применимое для веб-проектов — предоставляя свой собственный язык разметки XAML. Это позволяет создавать инструменты для разработки сравни тем, что применяются для разработки веб-сайтов и вовлекать дизайнеров в большей степени в процесс разработки приложения.

У разработчиков приложений под Windows времен, предшествующих появлению WPF, были все необходимые ингредиенты для разработки внешне привлекательных приложений, но также присутствовала проблема объединения их вместе. Большинство проблем, связанных с визуализацией уже были решены до появления WPF. Если у вас появится желание использовать всю мощь HTML и CSS в приложении под Windows, вас никто не остановит. Хотите масштабируемую и анимируемую графику, JavaScript предоставляет множество библиотек с различными видами анимации, и данную технологию также можно использовать в контексте Windows-приложения. Если вам не нравится то, как JavaScript реализует элементы управления и средства для взаимодействия

с ними, вы можете использовать Windows API. Стандартные элементы управления, предоставляемые Windows API, не могут похвастаться большим разнообразием внешнего вида, но они поддерживают всю необходимую функциональность, а также соответствуют всем стандартам в плане возможностей доступа (*accessibility*), и пользователи знают, как с ними взаимодействовать. Если же вы хотите задействовать всю мощь аппаратных средств, то вы можете использовать DirectX или OpenGL.

Проблема возникает в тот момент, когда у вас появляется необходимость в комбинировании возможностей этих или других технологий. Если вы хотите получить внешний вид элементов управления, создаваемый посредством JavaScript, но при этом сохранить для пользователей удобство использования элементов управления Windows API, у вас не получится этого добиться. Хотите поместить кнопку из Windows API внутрь 3D-сцены, созданной посредством DirectX? Это тоже не получится. Хотите применить анимацию, описанную при помощи CSS к элементам 3D-сцены? Не повезло. Корень проблемы заключается в том, что все эти технологии изолированы друг от друга. Вы можете их использовать бок о бок в пределах одного приложения, но при этом придется разделять пространство, отведенное под пользовательский интерфейс на части, и использовать в каждой из них какую-то конкретную технологию таким образом, чтобы эта часть не перекрывалась другой. При этом у вас, все равно, не получится применять эффекты или свойства одной технологии к элементам другой именно из-за отсутствия интеграции между ними.

WPF не приносит много новых возможностей из области визуализации. Большинство отдельно взятых вещей, которые умеет делать WPF, были доступны ранее в какой-то другой технологии. Уникальный вклад WPF в решение описанной проблемы именно в интеграции. Все описанные возможности доступны и при этом связаны вместе в пределах одной платформы. Это означает, что, если вы захотите поместить кнопку на грань трехмерной фигуры, вы сможете это сделать. Если вы захотите анимировать текст внутри поля для ввода, это возможно. Хотите создать совершенно новый внешний вид элемента управления, используя 2D- или 3D-графику и получить при этом полную поддержку средств ввода и функциональность аналогичную той, что есть в Windows API, и это возможно тоже.

3. Архитектура WPF

Как уже упоминалось раньше, основной программный интерфейс WPF, с которым должен взаимодействовать разработчик, предоставляется в виде управляемого кода. На ранних этапах разработки WPF было много споров по поводу того, где должна быть проведена черта, разделяющая управляемые и неуправляемые компоненты системы. CLR предоставляет много возможностей, которые заметно упрощают процесс разработки и делают его более продуктивным (управление памятью, система исключений, общая система типов и т.д.), но при этом у них есть своя цена.

На рис. 1 проиллюстрированы главные компоненты, входящие в состав или используемые WPF. Компоненты,

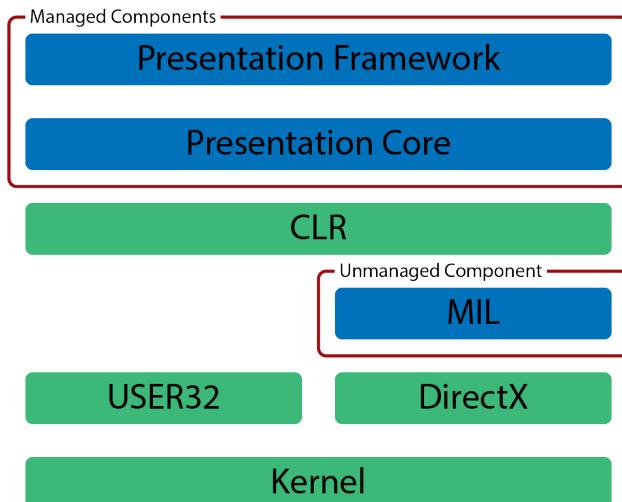


Рис. 1. Архитектура WPF

обозначенные синим цветом, являются частью WPF, а зеленым — частью операционной системы. Также показано, какие из компонентов WPF являются управляемыми, а какие нет.

В большинстве описаний WPF можно встретить упоминание о DirectX и какую большую значимость эта технология представляет. Стоит прояснить, что DirectX не является частью WPF, но используется им для отображения всей графики описанной в приложении. Для большего уточнения стоит отметить, что используется часть API под названием Direct3D, которая, как должно быть понятно из названия, отвечает за работу с трехмерной графикой. Несмотря на то, что в WPF возможна работа с трехмерной графикой, акцентировать свое внимание только на этом, значит упускать суть всей мощи технологии. WPF использует аппаратные средства, отвечающие за работу с трехмерной графикой даже при отображении двумерной графики пользователяского интерфейса. Это связано с тем, что все мониторы, которые существуют на сегодняшний день, отображают абсолютно все в виде плоской (двумерной) картинки. При выводе трехмерной графики, все трехмерные модели должны быть представлены в какой-то момент именно в двумерном виде (спроектированы), чтобы их можно было отобразить на плоском экране монитора. WPF взаимодействует именно с этой частью графического конвейера Direct3D, после того как прошло «уплощение» трехмерной сцены.

Непосредственно над DirectX располагается самый низкоуровневый компонент WPF — уровень медиа интеграции **Media Integration Layer** (или MIL), который

находится в **milcore.dll**. Этот уровень называется так, потому что он способен получать разные виды информации (битовые карты, видео, масштабируемая графика, текст и т.д.) и выводить их все на одну поверхность рисования DirectX. Это и есть то место, в котором происходит визуальная часть интеграции, о которой шла речь раньше. Стоит отметить, что MIL создан в виде неуправляемого кода. Это может оказаться неожиданным, учитывая тот факт, что WPF является частью .NET Framework. Причиной такого решения было то, что MIL взаимодействует напрямую с API DirectX, что выливается во множество COM-вызовов. COM — это технология, которая применена в DirectX, и понимание принципов ее работы не является необходимостью для разработки WPF-приложений, поэтому она не будет рассматриваться здесь. Если использовать COM-технологию напрямую из неуправляемого кода, то эти вызовы обходятся дешево для системы. Однако, обращение к COM из .NET Framework требует задействования слоя функциональной совместимости (*interoperability layer*), что делает эти вызовы существенно «дороже». Исходя из этого можно сказать, что основная цель MIL уменьшить накладные расходы, предоставив высокуровневый API для доступа из .NET Framework. API для MIL является недокументированным и не должен использоваться напрямую.

Над MIL находятся два другие компонента WPF: Presentation Core и Presentation Framework, расположенные в **PresentationCore.dll** и **PresentationFramework.dll** соответственно. Presentation Core, по сути, является открытым интерфейсом для MIL, который предоставляет

API в среде .NET Framework для взаимодействия с сервисами отображения описанными в MIL. Presentation Framework в свою очередь предоставляет более высокоуровневые сервисы, такие как: элементы управления, шаблоны, команды, привязка данных и др. Разработчики WPF-приложений большую часть времени взаимодействуют именно с компонентом Presentation Framework.

4. Обзор WPF

WPF обладает довольно крутой кривой обучаемости, и одной из самых больших проблем тех, кто начинает изучать данную технологию заключается в том, что понимание работы отдельных ее частей приходит с пониманием того, как эти части объединены вместе и какую роль играют во всей программной платформе. По этой причине, перед тем как начать детальное рассмотрение каждой части WPF, стоит рассмотреть вкратце несколько самых основных из них.

4.1. XAML

Одной из первых вещей, с которой вы столкнетесь работая с WPF, является язык разметки XAML (*Extensible Application Markup Language*). XAML не только самая известная особенность WPF, но и также одна из самых неправильно воспринимаемых. В общем и целом, если кто-то слышал о WPF, то он слышал и о XAML. При этом есть два не слишком широко известных факта о XAML. Во-первых, WPF не нуждается в XAML, и, во-вторых, XAML не нуждается в WPF. Это две отдельные технологии. Чтобы понять почему это так, давайте взглянем на фрагмент XAML-разметки:

XAML

```
<Button x:Name="submitButton" FontFamily="Consolas"  
FontSize="20" Foreground="Green">  
    Submit  
</Button>
```

В данном фрагменте разметки атрибуты описывают характеристики кнопки, такие как шрифт и цвет, а также описано ее содержимое (текст на кнопке). В результате на экране будет отображена кнопка (рис. 2).



Рис. 2. Кнопка

Из данного примера можно сделать вывод, что при помощи XAML описывается интерфейс приложения. Но как же в таком случае получается, что XAML не является неотъемлемой частью WPF? На самом деле XAML является не более чем языком для описания объектов .NET Framework. Для того, чтобы стало понятнее, о чем идет речь, давайте взглянем на программный код, который описывает аналогичную кнопку:

C#

```
var submitButton = new Button
{
    Content = "Submit",
    FontFamily = new FontFamily("Consolas"),
    FontSize = 20.0,
    Foreground = Brushes.Green
};
```

Создание элемента `<Button>` в XAML означает, что необходимо создать новый объект типа `Button`. Атрибут `x:Name` описывает имя, которое необходимо использовать для этого объекта, чтобы к нему можно было обращаться в коде. Атрибуты `FontFamily`, `FontSize` и `Foreground` относятся к соответствующим свойствам объекта, а содержи-

мое элемента относится к еще одному свойству Content. В этом и заключается практически весь смысл XAML. Элементы описывают объекты, а атрибуты описывают их соответствующие свойства. Конечно, при этом существует много всяческих деталей, как например, откуда XAML знает, что значение для атрибута **FontSize** должно быть числовым, в то время, как для атрибута **Foreground** — кистью. Также есть детали, описывающие то, каким образом XAML понимает, что делать с содержимым элемента, как в простых случаях, типа вышеописанной кнопки, так и в более сложных, как, например, для элемента управления список, который может содержать более одного элемента в качестве содержимого. Существует много подобных правил, и они будут рассмотрены детально позже в наиболее подходящих для них разделах. При этом самое важное, что вам стоит сейчас понять на счет XAML это то, что он описывает создание объектов и присваивание значений их свойствам.

Следующая не менее важная вещь, которую стоит запомнить это то, что все, что можно сделать, используя XAML, можно сделать, используя программный код, и стоит хорошо подумать перед тем как сделать выбор в пользу использования одного или другого. Довольно часто, те, кто только начинают изучать XAML, стараются делать все при помощи него, потому что, не смотря на его простоту, XAML является очень мощным инструментов для управления WPF. Совет в данном случае такой же и как в большинстве подобных ситуаций в программировании: если у вас есть несколько инструментов, при помощи которых можно решить поставленную перед

вами задачу, вы должны решать ее при помощи наиболее подходящего из них. Несмотря на то, что XAML является действительно великолепной технологией, он не всегда является лучшим способом решения задачи.

Вот, что имеется в виду, когда говорится, что WPF не нуждается в XAML. XAML не позволяет делать ничего такого, что бы нельзя было сделать в коде. А что же на счет обратного утверждения? Также было сказано, что XAML не нуждается в WPF. Скорее всего, ответ на этот вопрос уже очевиден. Если XAML является просто языком для описания объектов, то ему не особо важно знать, что именно это за объекты. Он использует простые правила на основании пространств имен XML, чтобы определить какой именно тип данных необходимо использовать для каждого элемента.

В фрагменте разметки рассмотренной ранее не видно, но элемент описывающий кнопку, требует пространство имен по умолчанию, чтобы эта запись имела смысл. Есть пространство имен, содержащее все элементы, которые чаще всего используются при написании XAML-разметки для WPF-приложений. Также существуют и другие пространства имен, например, для Silverlight (еще одна технология от Microsoft), которая использует тот же XAML, но с другим набором элементов. А еще есть пространство имен для технологии Windows Workflow Foundation, которая использует XAML для описания рабочих процессов, что и близко не похоже на описание пользовательских интерфейсов. И, конечно же, вы можете создавать собственные пространства имен, которые будут содержать ваши пользовательские типы.

При большом желании можно даже научить XAML создавать интерфейсы для приложений Windows Forms. Однако, если вы это попробуете, то быстро обнаружите, что не все API будет одинаково удобно контролировать из XAML. Описание интерфейса приложения Windows Forms посредством XAML не выглядит столь же элегантным как использование XAML в WPF. Это связано с тем, что WPF проектировался с мыслью о XAML. Для каждого свойства WPF, разработчики из Microsoft потратили много времени и усилий, чтобы учесть то, как оно будет описываться в XAML. При этом был создан API, который работает одинаково хорошо, как из кода, так и из разметки. Несмотря на то, что XAML является опциональным, он лежит в основе дизайна WPF.

Целью использования XAML есть вовлечение дизайнёров в процесс разработки пользовательского интерфейса напрямую. Поэтому WPF спроектирован таким образом, чтобы можно было управлять всеми аспектами визуальной составляющей приложения при помощи разметки.

4.2. Деревья пользовательского интерфейса

При помощи XAML описывается пользовательский интерфейс приложения, образуя древовидную структуру объектов. Пример с кнопкой, содержащий текстовый узел в качестве дочернего элемента, описанный ранее был очень тривиальным. Интерфейсы приложений в подавляющем большинстве случаев выглядят гораздо сложнее, и древовидная структура в них заметна куда лучше. В качестве примера давайте рассмотрим другой фрагмент XAML-разметки:

XAML

```
<Page>
    <StackPanel Orientation="Horizontal">
        <Label>Message:</Label>
        <TextBox Width="100"/>
    </StackPanel>
</Page>
```

Корневым элементом является страница (`<Page>`), содержащая единственный дочерний элемент — панель (`<StackPanel>`), задачей которой является определение расположения элементов, помещенных в нее. В данном примере она организовывает элементы метку (`<Label>`) и поле для ввода текста (`<TextBox>`) в горизонтальный стек (рис. 3).



Рис. 3. Метка и поле для ввода текста

Пример, по-прежнему, довольно простой, но, тем не менее, древовидная структура здесь нагляднее: корневой узел содержит один дочерний узел, который содержит еще несколько дочерних узлов. Однако, если учесть все, что показывается (или может быть показано) на экране, то получится гораздо больше четырех узлов, описанных в разметке. Элементы страницы и панели не видны здесь, потому что они предназначены для организации содержимого и не имеют визуального представления. Если рассмотреть поле для ввода текста, расположенное справа, то можно заметить, что этот элемент состоит из нескольких частей. У него есть контур, мигающий курсор,

если бы был введен текст, то он тоже отображался бы, а еще у текста может быть выделение и полосы прокрутки. Скромное поле для ввода текста, как оказывается, состоит из множества отдельных частей. Имеет смысл рассматривать поле для ввода текста как один цельный элемент, но также не менее обосновано его можно считать сущностью, состоящей из нескольких частей. Следовательно, в WPF существует два вида представления структуры пользовательского интерфейса: логическое дерево, как показано в примере разметки выше, и визуальное дерево, которое содержит все элементы необходимые для отображения пользовательского интерфейса (рис. 4).

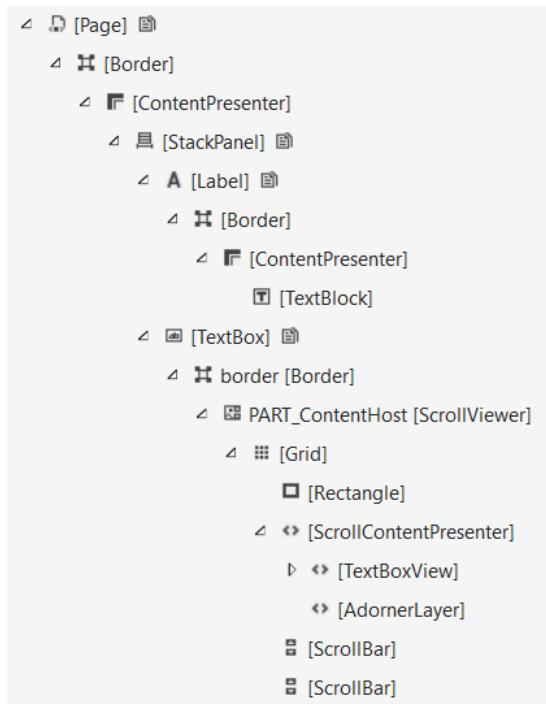


Рис. 4. Визуальное дерево

Визуальное дерево является надмножеством логического дерева, и если посмотреть на них, то можно заметить, что все логические элементы также присутствуют в визуальном дереве. Таким образом, оба эти дерева являются всего лишь разными представлениями одного и того же набора элементов, из которых состоит интерфейс приложения. При этом логическое дерево не включает много деталей, что позволяет сконцентрироваться на основной структуре пользовательского интерфейса и игнорировать детали того, как именно она представляется на экране. Именно логическое дерево используется разработчиками при проектировании основного каркаса интерфейса приложения. Визуальное дерево представляет интерес в тех ситуациях, в которых вам необходимо сосредоточиться на представлении, например, изменить внешний вид поля для ввода текста.

4.3. События и команды

Большинству приложений необходимо взаимодействовать с пользователем, а именно, реагировать на устройства ввода тем или иным образом. Для этих целей в WPF существует две конструкции: события и команды. События тесно связаны с визуальным деревом. Когда пользователь для ввода использует клавиатуру или мышь, соответствующие события возникают в конкретном элементе пользовательского интерфейса, в том у которого фокус клавиатурного ввода, или в том который находится под курсором мыши. Как вы могли заметить ранее, визуальное дерево может быть до-

вольно сложным и вам вряд ли захочется прикреплять обработчик события к каждому элементу визуального дерева только для того, чтобы гарантировать получение ими ввода. WPF предоставляет механизм маршрутизации событий для решения подобных проблем. Этот механизм позволяет обрабатывать события предками элемента, для которого возникло событие изначально. Существует два вида маршрутизации: туннелирование (*tunneling*) и всплытие (*bubbling*).

При туннелировании, событие начинается с корневого элемента визуального дерева и направляется по соответствующим дочерним элементам вплоть до того, для которого событие предназначалось изначально (рис. 5).

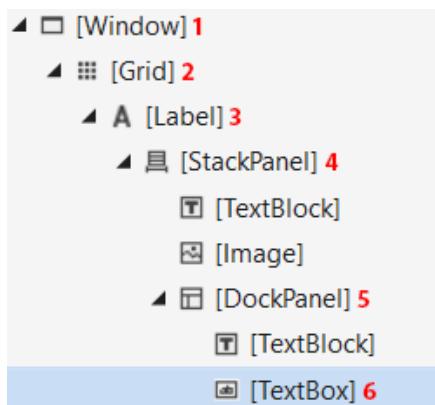


Рис. 5. Туннелирование события

Всплытие события, наоборот, начинается с элемента, для которого оно предназначено и продвигается вверх по дереву до тех пор, пока не будет найден элемент, который может обработать событие (рис. 6).

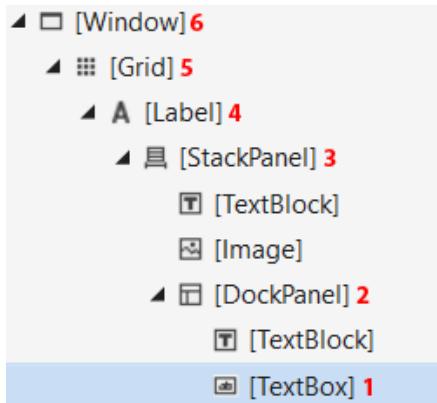


Рис. 6. Всплытие события

События в WPF обычно существуют попарно. Есть предварительное событие (*preview*), которое туннелируется, и есть основное событие, которое всплывает. Идея предварительных событий заключается в том, чтобы дать возможность родительским элементам отреагировать на событие раньше. Например, окно может реагировать на нажатие кнопки мыши, до того, как это сделает элемент управления, на котором было нажатие.

Как правило, событие связано с более низкоуровневым взаимодействием, таким как нажатие кнопок мыши или нажатие клавиш на клавиатуре, но, часто, разработка приложения требует более высокого уровня абстракции. Например, вам может быть абсолютно все равно, была ли нажата комбинация горячих клавиш **Ctrl+P** или была нажата кнопки печати на панели инструментов или был выбран пункт меню «Печать...», потому что во всех этих ситуациях вы хотите, чтобы приложение реагировало одинаково — был начат процесс печати. WPF предоставляет механизм команд, который позволяет получить подоб-

ный уровень абстракции. Вы можете написать команду печати в виде отдельного обработчика, и ассоциировать ее с необходимой комбинацией горячих клавиш, кнопкой или пунктом меню. Подробный процесс взаимодействия с событиями и командами будет рассмотрен позже.

4.4. Элементы управления

WPF поддерживает концепцию элементов управления (*control*), но означает она здесь не совсем то же самое, что и в более ранних технологиях Microsoft для представления графического пользовательского интерфейса, как, например, Windows Forms. В этих, более ранних технологиях, элемент управления представляет из себя видимый объект, что-то, что имеет внешний вид, с которым может взаимодействовать пользователь, а также API для разработчиков. Элементы управления в WPF похожи на это, но не полностью.

Давайте рассмотрим в качестве примера кнопку. Кнопка предоставляет событие нажатия, что можно отнести к API, а также к взаимодействию с пользователем. Она может получать фокус, что можно отнести к характеристике интерактивности. С ней можно связать команду, что опять же является частью API. Все это, как и ряд других, не упомянутых здесь, особенностей поддерживается в WPF также, как это присутствовало в более ранних аналогичных технологиях в том или ином виде. Но есть одна очень важная характеристика, которой нет у кнопки в WPF, а именно — внешнего вида. Она не знает, как именно она должна себя отображать на экране. И это главное отличие элементов управления в WPF от своих

предшественников. В компании Microsoft даже придумали специальное слово, описывающее эту особенность элементов управления — lookless. К сожалению, точного перевода это слово не имеет, но, примерно, его можно перевести как «не имеющий внешнего вида». При этом, внешний вид элементам управления задается при помощи отдельной сущности — шаблона. Одному и тому же элементу управления может требоваться разный внешний вид в зависимости от контекста его использования (рис. 7).

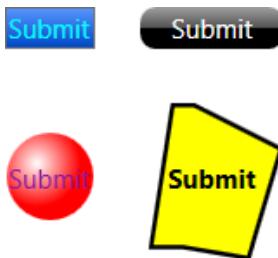


Рис. 7. Применение разных шаблонов к одной кнопке

Кнопка, как и другие элементы не должна знать о том, как она выглядит, потому что разные темы оформления могут требовать разные стили визуализации или приложение может обладать своим собственным уникальным стилем. Именно по этой причине элементы управления в WPF разделены на две части: внешний вид определяется шаблоном, а все остальное — самим элементом управления.

4.5. Примитивные элементы

Раз уж элементы управления не определяют то, как они выглядят, тогда каким образом хоть что-то может быть показано на экран? Может показаться,

что внешний вид, скажем, элемента прямоугольник определяется его шаблоном, который в свою очередь будет содержать элемент прямоугольник, шаблон которого будет использоваться для его отображения и т.д., что в итоге вылилось бы в бесконечную рекурсию. Но такая проблема отсутствует, потому что большинство элементов, которые могут отображаться на экране, не являются элементами управления. От базового класса, описывающего элемент управления (`System.Windows.Controls.Control`) наследуются только те классы, которые описывают какой-либо вид интерактивного поведения, такие как: кнопки, списки, поля для ввода текста, полосы прокрутки и т.д. Это все элементы управления. Однако, классы, описывающие прямоугольник, эллипс, изображение и др. не наследуются от базового класса элемента управления. Они примитивнее и наследуются от более базового класса `System.Windows.FrameworkElement`, который также является базовым для класса `System.Windows.Controls.Control`. На самом деле этот класс является базовым практически для всего, что находится в визуальном дереве. Только наследники класса `System.Windows.Controls.Control` содержат шаблон, описывающий их визуализацию, о котором шла речь ранее. Наследники же класса `System.Windows.FrameworkElement` не имеют шаблона, но при этом они обладают собственной визуализацией. Другим отличием можно считать то, что элементы управления, обычно, имеют какое-то базовое поведение, а примитивные элементы — нет. Например, поле для ввода текста `System.Windows.Controls.TextBox` поддерживает ввод и редактирование текста, в то время,

как примитивный элемент `System.Windows.Controls.TextBlock` умеет только отображать текст.

Многие называют элементом управления все, что может находиться в визуальном дереве, включая даже то, что не наследуется от класса `System.Windows.Controls.Control`. В определенном смысле это обосновано. Пространство имен `System.Windows.Controls` содержит много всего, что не наследуется от этого класса. Для того, чтобы избежать путаницы в терминологии стоит немного подытожить. Элемент управления — это тип данных, который наследуется от класса `System.Windows.Controls.Control` и является элементом без внешнего вида, но поддерживающий работу с шаблонами. Примитивный элемент — это тип данных, который наследуется от класса `System.Windows.FrameworkElement` и обладает своим собственным внешним видом, и не поддерживает работу с шаблонами. Если же речь идет просто об элементе из визуального дерева, то это может быть любой из последних перечисленных.

4.6. Панели

Одной из наиболее полезных особенностей WPF является набор панелей. Панель представляет из себя элемент, который можно добавить в визуальное дерево, и который может содержать несколько дочерних элементов. Задачей любой панели является определение местоположения каждого из дочерних элементов согласно присущей ей логике. При решении любой задачи компоновки нескольких элементов, задействуются панели, по отдельности или вместе.

Существует несколько видов панелей, разнящихся от довольно простых, позволяющих позиционировать элементы по точно заданным координатам, или выстраивать их в вертикальный или горизонтальный стек, до более сложных, позволяющих компоновать элементы в виде сетки. Более подробно каждая из панелей будет рассмотрена позже. На данном этапе, ключевым моментом, который стоит запомнить является то, что если вам необходимо в каком-либо месте визуального дерева расположить несколько элементов, то, практически всегда, будут задействоваться панели.

4.7. Документы нефиксированного формата

Когда речь идет об элементе на подобие кнопки, то, обычно, имеет значение где именно она будет располагаться в пределах интерфейса, в верхнем левом углу, правом нижнем углу и т.д. Но когда речь идет о словах в фрагменте текста, то расположение каждого отдельного слова, обычно, не имеет значения. Имеет значение их порядок и то, чтобы они формировали строки удобочитаемой длины. Авторам текста редко важно где именно будет находиться конкретное слово, вверху или внизу страницы, и точное расположение изображения не всегда имеет значение, главное, чтобы оно появилось рядом с текстом, к которому оно имеет отношение. Точное позиционирование таких элементов чаще всего является контрпродуктивным, так как намного полезнее иметь возможность адаптировать текст под доступное пространство, в зависимости от ситуации.

WPF предоставляет средства для формирования документов, которые могут подстраивать свое содержимое

под доступное им пространство, с учетом заданных правил форматирования. Несмотря на то, что такой подход к формированию интерфейса сильно отличается от того, что рассматривалось ранее, оба варианта можно объединить вместе. На рис. 8 вы можете увидеть, как кнопка и поле для ввода текста располагаются в пределах текста.

Flow documents are designed to optimize viewing and readability. Rather than being set to one predefined layout, flow documents dynamically adjust and reflow their content based on run-time variables such as window size, device resolution, and optional user preferences. In addition, flow documents offer advanced document features, such as pagination and columns. This topic provides an overview of flow documents and how to create them.

appearance of fonts. Flow Documents are best utilized when ease of reading is the primary document consumption scenario. In contrast, Fixed Documents are designed to have a static presentation. Fixed Documents are useful when fidelity of the source content is essential. See Documents in WPF for more information on different types of documents.



A flow document is designed to "reflow content" depending on window size, device resolution, and other environment variables. In addition, flow documents have a number of built in features including search, viewing modes that optimize readability, and the ability to change the size and

Рис. 8. Документ нефиксированного формата

Т.е., вполне реально интегрировать текст в пользовательский интерфейс, точно так же, как и реально интегрировать элементы пользовательского интерфейса в текст.

Возможности по работе с текстом предоставляемые разработчикам WPF- приложений похожи на те, что предоставляет HTML. Документы могут использовать разные виды форматирования текста, таблицы, нумерованные и ненумерованные списки и много чего другого. Также WPF предоставляет возможности разбиения содержимого на страницы и колонки, чтобы лучше адаптироваться под размеры доступной рабочей области и повысить читабельность текста.

5. Обзор XAML

Как упоминалось ранее, XAML является отдельной от WPF технологией. Вся информация, касающаяся данной технологии, которая будет рассматриваться далее, будет касаться XAML в виде, реализованном в WPF.

Использование XAML позволяет описать элементы пользовательского интерфейса при помощи разметки, поместив описание их поведения в отдельные файлы с программным кодом, так называемая, модель *code-behind*, которая детально будет рассмотрена далее. В отличие от большинства языков разметки, XAML напрямую описывает создание объектов, тем самым имея большую привязку к лежащей в основе системе типов. Использование XAML позволяет добиться рабочего процесса, в котором разные группы разработчиков могут отдельно работать над созданием пользовательского интерфейса и логики приложения. При этом могут даже использоваться совершенно разные инструменты разработки.

Как правило, XAML содержится в текстовых файлах, обладающих расширением `.xaml`. Данные файлы могут содержать информацию в виде любой кодировки, но, чаще всего, используется кодировка UTF-8.

5.1. Синтаксис XAML

В данном разделе будет рассмотрен синтаксис XAML, большая часть которого может показаться знакомым тем, кто знает XML. XAML определяет ряд собственных

концепций, которых нет в XML, но при этом они вписываются в его синтаксис.

Большинство примеров, приведенных в этом разделе, являются лишь фрагментами полной разметки и не будут работать сами по себе. Это сделано для того, чтобы при объяснении какой-то конкретной особенности синтаксиса можно было сконцентрировать внимание только на ней, и не отвлекаться на большие фрагменты сопутствующей разметки, необходимой для полноценного функционирования. К тому же, в большинстве случаев, она имеет одну и ту же структуру. То же самое касается приведенного здесь программного кода. Далее будут рассмотрены примеры, в которых будет показано, как отдельные части соединяются вместе.

Чувствительность к регистру

XAML является чувствительным к регистру языком, как и XML. Это связано с тем, что элементам, описанным в разметке, соответствуют программные типы данных, которые также чувствительны к регистру.

Элемент

Элементы в XAML предназначены для описания экземпляров объектов. Это означает, что после того как разметка будет обработана XAML-анализатором, будут созданы объекты соответствующих типов. Немаловажной особенностью является то, что для их создания будет использован конструктор типа данных по умолчанию. Синтаксис описания элемента выглядит следующим образом:

XAML

```
<TypeName>Content</TypeName>
```

Описание элемента состоит из открывающего (`<TypeName>`) и закрывающего (`</TypeName>`) тегов, между которыми располагается содержимое (*Content*). Здесь TypeName означает имя элемента, которое, по сути, является названием типа данных. Имя может обладать дополнительным префиксом. Также, после имени,optionally, могут располагаться атрибуты. Смысл и синтаксис префиксов имен и атрибутов будут показаны далее.

В качестве содержимого, теоретически (согласно правил синтаксиса языка), может выступать как обычный текст, так и другие элементы (которые называются дочерними) в любой комбинации. На практике же, содержимое каждого элемента строго описано типом данных, к которому он относится. Некоторые могут иметь только один дочерний элемент, а некоторые более одного. Другие могут содержать только текст, в качестве содержимого. А бывает, что содержимого вообще не должно быть. Что именно может или должен содержать элемент будет оговариваться детально при рассмотрении каждого конкретного типа данных.

Как упоминалось ранее, бывает, что у элемента в определенном контексте нет содержимого или он, в принципе, не может им обладать. В этой ситуации, между открывающим и закрывающим тегом ничего не пишется:

XAML

```
<TypeName></TypeName>
```

В таком случае можно воспользоваться второй формой описания элемента, которая предназначена специально для описания подобных ситуаций. В этом варианте син-

таксиса полностью отсутствует описание содержимого и закрывающего тега:

XAML

```
<TypeName />
```

Вместо открывающего и закрывающего тегов здесь присутствует, так называемый, самозакрывающийся тег (`<TypeName/>`).

Стоит отметить, что обе формы являются корректными с точки зрения синтаксиса языка и могут использоваться равноправно.

Давайте рассмотрим пример:

XAML

```
<Grid>
    <TextBox/>
</Grid>
```

Данный фрагмент разметки демонстрирует создание двух элементов: `<Grid>`, у которого есть содержимое и закрывающий тег, и `<TextBox>`, который использует самозакрывающуюся форму. Фактически, здесь происходит создание двух объектов, обладающих типами `System.Windows.Controls.Grid` и `System.Windows.Controls.TextBox` соответственно:

C#

```
var textBox = new TextBox();
var grid = new Grid();
grid.Children.Add(textBox);
```

Атрибут

Атрибуты элементов используются для того, чтобы задавать значения свойств соответствующих объектов. Синтаксис описания атрибутов выглядит следующим образом:

XAML

```
<TypeName PropertyName="Value">Content</TypeName>
<TypeName PropertyName="Value"/>
```

Если требуется задать значения для более чем одного атрибута, в таком случае атрибуты описываются друг за другом и разделяются пробелом. Порядок атрибутов не имеет значения.

XAML

```
<TypeName PropertyName1="Value1" PropertyName2="Value2"/>
```

В качестве примера давайте рассмотрим следующий фрагмент разметки:

XAML

```
<Button Height="30" Width="100">Yes</Button>
```

Этой разметке будет соответствовать следующий программный код:

C#

```
var button = new Button { Content = "Yes",
Height = 30.0, Width = 100.0 };
```

Синтаксис атрибутов также может быть использован для указания обработчиков событий:

XAML

```
<TypeName EventName="HandlerName"/>
```

Где EventName имя события, а HandlerName — название метода, который является обработчиком данного события. В качестве примера давайте рассмотрим кнопку и событие щелчка:

XAML

```
<Button Click="Button_Click">OK</Button>
```

Подробнее о свойствах и где именно должен находиться метод Button_Click будет рассказано далее.

Свойство можно описывать не более одного раза для каждого объекта. Следующая разметка генерирует ошибку:

XAML

```
<Button Height="100" Height="50">OK</Button>
```

Элемент-свойство

Как бы ни была удобна форма присваивания строк в качестве значений, существует множество ситуаций, где такой вариант будет неудобен или даже невозможен. Например, значением свойства является другой объект, у которого есть целый ряд своих свойств, требующих присваивания им значений. К счастью, в XAML предусмотрена альтернативная запись для присваивания значений свойствам объекта, так называемый синтаксис «элемент-свойство». Для того, чтобы описать какое-то свойство объекта в такой форме, необходимо добавить дочерний элемент в тот,

который требует описания свойства. Имя этого дочернего элемента должно быть устроено по следующему принципу. Как обычно, знак меньше (<), потом название родительского элемента, далее точка (.), за которой следует название свойства, и завершает последовательность знак больше (>):

XAML

```
<TypeName>
    <TypeName.PropertyName>
        <AnotherTypeName AnotherPropertyName="Value"/>
    </TypeName.PropertyName>
</TypeName>
```

В качестве примера давайте посмотрим, как можно описать значение для свойства, отвечающего за задний фон поля для ввода текста.

XAML

```
<TextBox FontFamily="Consolas" FontSize="20"
Foreground="Yellow">
    <TextBox.Background>
        <SolidColorBrush Color="Aqua" Opacity="0.5"/>
    </TextBox.Background>
</TextBox>
```

Данной разметке будет соответствовать следующий программный код:

C#

```
var textBox = new TextBox
{
    Background = new SolidColorBrush { Color =
        Colors.Aqua },
    FontFamily = new FontFamily("Consolas"),
```

```
    FontSize = 20.0,  
    Foreground = Brushes.Yellow  
};
```

Подобная форма записи может использоваться и для обычных строковых значений. Ниже показаны две формы присваивания значения отвечающего за размер шрифта. Оба варианта делают абсолютно идентичные вещи.

XAML

```
<TextBox FontSize="20"/>  
<TextBox>  
    <TextBox.FontSize>20</TextBox.FontSize>  
</TextBox>
```

При необходимости, можно делать более одного вложенного описания для разных свойств одного объекта. Также возможно комбинировать их вместе с описанием обычных атрибутов.

XAML

```
<TextBox Foreground="Red">  
    <TextBox.FontFamily>Consolas</TextBox.FontFamily>  
    <TextBox.FontSize>20</TextBox.FontSize>  
</TextBox>
```

Нельзя описывать одно и то же свойство для одного объекта, применяя обе формы одновременно. Следующая разметка генерирует ошибку:

XAML

```
<TextBox Foreground="Red">  
    <TextBox.Foreground>Red</TextBox.Foreground>  
</TextBox>
```

Свойство-коллекция

XAML включает в себя ряд оптимизаций, которые призваны создать более читабельную разметку. Одной из таких оптимизаций является то, как в XAML присваиваются значения свойствам, которые работают с коллекциями объектов. Синтаксис описания свойств-коллекций:

XAML

```
<TypeName>
    <TypeName.CollectionPropertyName>
        <CollectionName>
            <Item1/>
            <Item2/>
            <Item3/>
        </CollectionName>
    </TypeName.CollectionPropertyName>
</TypeName>
```

Существует упрощенный вариант этой записи, который делает практически то же самое, и, который применяется гораздо чаще. В этом случае можно не указывать элемент коллекции явным образом.

XAML

```
<TypeName>
    <TypeName.CollectionPropertyName>
        <Item1/>
        <Item2/>
        <Item3/>
    </TypeName.CollectionPropertyName>
</TypeName>
```

В примере приведенном ниже показано, как присваивается коллекция значений свойству System.Windows.Media.GradientBrush.GradientStops.

XAML

```
<TextBox FontFamily="Consolas" FontSize="20"
        Foreground="Yellow">
    <TextBox.Background>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Color=
                        "Red" Offset="0.0"/>
                    <GradientStop Color=
                        "Pink" Offset="1.0"/>
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </TextBox.Background>
</TextBox>
```

Здесь создается объект коллекции System.Windows.Media.GradientStopCollection при помощи конструктора по умолчанию, после чего в него добавляются объекты System.Windows.Media.GradientStop, при помощи метода System.Windows.Media.GradientStopCollection.Add. Далее объект коллекции присваивается свойству System.Windows.Media.GradientBrush.GradientStops. Этую же разметку можно записать и по-другому:

XAML

```
<TextBox FontFamily="Consolas"
        FontSize="20" Foreground="Yellow">
    <TextBox.Background>
```

```
<LinearGradientBrush>
    <LinearGradientBrush.GradientStops>
        <GradientStop Color="Red" Offset="0.0"/>
        <GradientStop Color="Pink" Offset="1.0"/>
    </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</TextBox.Background>
</TextBox>
```

Этот случай немного отличается от предыдущего. Здесь для свойства `System.Windows.Media.GradientBrush.GradientStops` вызывается `get`-метод, который возвращает текущий объект коллекции, после чего в него добавляются объекты `System.Windows.Media.GradientStop`, при помощи метода `System.Windows.Media.GradientStopCollection.Add`.

Может показаться, что эти отличия не существенны и можно не обращать на них внимания, но это не так. Например, в первом случае элементы `System.Windows.Media.GradientStop` добавляются в объект, который был создан заранее, а значит, там могут быть и другие элементы, которые не будут удалены. Во втором случае используется конструктор по умолчанию для создания объекта коллекции, который не всегда может быть доступным, что сделает этот вариант не пригодным для использования. Также, этот вариант требует использования `set`-метода свойства-коллекции.

Главным требованием со стороны кода для использования синтаксиса свойств-коллекций является следующее. Свойство-коллекция должно обладать типом данных, который реализует интерфейс `System.Collections.IList`.

Свойство «содержимое»

XAML предоставляет функциональность, при помощи которой каждый класс может обозначить одно любое свое свойство, как свойство, предназначенное для хранения содержимого. Содержимое (дочерний элемент или текст) элемента данного класса, будет автоматически присвоено в это свойство. Эта возможность позволяет получить более наглядную структуру родственных связей элементов. Чаще всего это свойство называется Content, но, в принципе, оно может называться, как угодно.

В качестве примера можно рассмотреть класс System.Windows.Controls.Border, в котором свойством содержимого является свойство System.Windows.Controls.Decorator.Child. Ниже показан фрагмент разметки, где создается два элемента по-разному, при этом, оба варианта полностью идентичны.

XAML

```
<Border>
    <Border.Child>
        <TextBlock Text="Hello"/>
    </Border.Child>
</Border>

<Border>
    <TextBlock Text="Hello"/>
</Border>
```

Согласно синтаксиса XAML, значение для свойства содержимого должно быть полностью описано либо до всех элементов-свойств, либо после всех, как это показано в примере ниже:

XAML

```
<Button>
    <Button.Background>Red</Button.Background>
    Some long content for button 1
</Button>
<Button>
    Some long content for button 2
    <Button.Background>Red</Button.Background>
</Button>
```

Следующий пример демонстрирует вариант неправильной записи, так как часть содержимого описана до элемента-свойства, а часть после:

XAML

```
<Button>
    Some long
    <Button.Background>Red</Button.Background>
    content for button 3
</Button>
```

Комбинация свойства «содержимое» и свойства-коллекции

Давайте взглянем на пример:

XAML

```
<StackPanel>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
</StackPanel>
```

В данном примере присваивается значение свойству содержимого элемента **<StackPanel>**, которое, к тому же,

является еще и свойством-коллекцией. В полной форме этот фрагмент кода выглядел бы примерно так:

XAML

```
<StackPanel>
    <StackPanel.Children>
        <!--<UIElementCollection>-->
        <Button>Button 1</Button>
        <Button>Button 2</Button>
        <!--</UIElementCollection>-->
    </StackPanel.Children>
</StackPanel>
```

Свойство `System.Windows.Controls.Panel.Children` является свойством содержимого класса `System.Windows.Controls.StackPanel`. Тип этого свойства — коллекция `System.Windows.Controls.UIElementCollection`. Однако у этой коллекции нет доступного конструктора по умолчанию, поэтому эта часть разметки закомментирована.

Присоединенное свойство

Механизм присоединенных свойств позволяет описать свойство в одном типе данных, и использовать его в качестве атрибутов для элемента совершенно другого типа. Такой механизм чаще всего применяется в ситуациях, когда дочерние элементы должны задать какие-то данные своему родительскому элементу, но, помимо этого, присоединенные свойства могут использоваться и в других ситуациях.

Синтаксис присоединенных свойств напоминает синтаксис элемента-свойства. Для того, чтобы описать такой атрибут, необходимо в качестве имени атрибута написать

название класса, в котором объявлено присоединенное свойство, далее поставить точку (.), и написать название свойства:

XAML

```
<TypeName AnotherTypeName.PropertyName="Value"/>
```

В качестве примера рассмотрим вариант разметки, в котором создается панель сетка (`<Grid>`), в которую помещается поле для ввода текста (`<TextBox>`). При этом, полю для ввода текста необходимо указать в какую именно ячейку сетки оно должно быть помещено. Это делается при помощи присоединенных свойств `System.Windows.Controls.Grid.Column` и `System.Windows.Controls.Grid.Row`.

XAML

```
<Grid>
    <TextBox Grid.Column="0" Grid.Row="0"/>
</Grid>
```

На самом деле, присоединенные свойства не являются свойствами, они являются статическими методами. Вот примерный программный код, который генерируется для разметки описанный выше:

C#

```
var textBox = new TextBox();

var grid = new Grid();
grid.Children.Add(textBox);

Grid.SetColumn(textBox, 0);
Grid.SetRow(textBox, 0);
```

Как видно, никаких свойств тут нет, лишь методы. Правило трансляции разметки присоединенных свойств в программный код довольно простое. Запись **Type***Name*.
PropertyName=*Value* транслируется в **Type***Name*.
SetPropertyName(obj, Value), при этом аргумент obj — это тот объект, для которого происходит присваивание значения присоединенному свойству, т.е. в примере выше это поле для ввода текста.

При необходимости вы можете создавать свои собственные присоединенные свойства и использовать их точно так же, как и те, что описаны в стандартных классах.

Присоединенные свойства можно также описывать, используя синтаксис элемента-свойства:

XAML

```
<Grid>
    <TextBox>
        <Grid.Column>0</Grid.Column>
        <Grid.Row>0</Grid.Row>
    </TextBox>
</Grid>
```

Нельзя одно и то же свойство описать для одного элемента дважды, даже используя две разные формы записи. Следующий пример вызовет ошибку, так как свойство **System.Windows.Controls.Grid.Column** описано дважды.

XAML

```
<Grid>
    <TextBox Grid.Column="0">
        <Grid.Column>0</Grid.Column>
        <Grid.Row>0</Grid.Row>
```

```
</TextBox>
</Grid>
```

Преобразователь типа

Давайте рассмотрим пример, в котором описывается элемент управления кнопка (`<Button>`), с указанием свойства, отвечающего за толщину контура — `System.Windows.Controls.Control.BorderThickness`.

XAML

```
<Button Content="Submit">
    <Button.BorderThickness>
        <Thickness Bottom="10" Left="15" Right="10"
Top="5"/>
    </Button.BorderThickness>
</Button>
```

Как видно из примера, свойству `System.Windows.Controls.Control.BorderThickness` присваивается объект `System.Windows.Thickness`, в свойства которого прописываются толщина каждой из частей рамки кнопки. Данная запись выглядит вполне логично, и, очевидно, что мы не смогли бы присвоить сразу четыре значения свойству `System.Windows.Controls.Control.BorderThickness`, не воспользовавшись элементом-свойством. Однако, есть альтернативная форма записи точно такой же разметки:

XAML

```
<Button BorderThickness="15,5,10,10" Content="Submit"/>
```

В данном случае, свойству `System.Windows.Controls.Control.BorderThickness`, которое обладает типом данных

System.Windows.Thickness была присвоена строка вида 15,5,10,10, и сработало это за счет того, что для класса System.Windows.Thickness существует, так называемый, преобразователь типа, который является классом производным от System.ComponentModel.TypeConverter. Это довольно простой класс, который содержит пару методов, для преобразования строки в необходимый тип данных и наоборот. Следовательно, если вы хотите, чтобы в XAML-разметке можно было присваивать строки определенного формата, которые бы автоматически преобразовывались в объекты требуемых типов, вам необходимо реализовать свой собственный преобразователь типа. Как это сделать будет подробнее рассмотрено далее.

В WPF существует много готовых преобразователей, для часто используемых типов данных, которые мы также будем детальнее рассматривать по мере необходимости.

Расширение разметки

Способы устанавливать значения в свойства объекта, рассмотренные ранее, отлично справляются с поставленной перед ними задачей в большинстве ситуаций. Хотя, у всех них есть один большой недостаток — они устанавливают фиксированное значение, которое жестко задается непосредственно в разметке. Это не всегда удобно или возможно. Есть множество ситуаций, когда значение должно быть получено динамически на этапе выполнения или необходимо выполнить определенный фрагмент кода, чтобы его вычислить. В обоих случаях обычные варианты установки значений не подойдут. Для решения такой проблемы были придуманы, так назы-

ваемые, расширения разметки XAML. Эти расширения позволяют анализатору XAML получать значения из стороннего класса, вместо непосредственно разметки.

Все расширения разметки реализованы в виде отдельных классов, которые наследуются от базового класса `System.Windows.Markup.MarkupExtension`. Этот класс очень простой — он содержит всего лишь один метод `ProvideValue`, предоставляющий необходимое значение. Другими словами, когда появляется необходимость в получении значения, создается экземпляр указанного класса расширения разметки и вызывается его метод `System.Windows.Markup.MarkupExtension.ProvideValue`, который возвращает требуемое значение. Существует несколько готовых расширений разметки, которые будут рассмотрены позже, а также, вы можете создавать собственные. Синтаксис использования расширения разметки следующий:

XAML

```
<TypeName PropertyName="{ExtensionName Argument}" />
```

Здесь `ExtensionName` задает название класса, описывающего желаемое расширение разметки, а `Argument` — аргумент, передаваемый в конструктор этого класса. Вот пример использования одного из стандартных расширений:

XAML

```
<Button Style="{StaticResource OkButtonStyle}">OK</Button>
```

Здесь применяется класс расширения разметки под названием `System.Windows.StaticResourceExtension`.

Принято называть классы расширений так, чтобы они заканчивались суффиксом `Extension`, но это не яв-

ляется синтаксическим ограничением. При этом, XAML позволяет не писать этот суффикс.

Синтаксис использования расширений разметки со стороны XAML описывает еще несколько возможностей, например, присваивание дополнительных свойств объекту класса расширения:

XAML

```
<TypeName PropertyName="{ExtensionName Argument,  
PropertyName=Value}" />
```

В случае, если класс расширения разметки содержит конструктор по умолчанию его также можно использовать, не указав аргумент:

XAML

```
<TypeName PropertyName="{ExtensionName  
PropertyName=Value}" />
```

Также можно использовать класс расширения разметки, не задав дополнительно ни одного свойства и указав использование конструктора по умолчанию:

XAML

```
<TypeName PropertyName="{ExtensionName}" />
```

Корневой элемент и пространства имен

Любой XAML-файл должен содержать один и только один корневой элемент, чтобы являться синтаксически правильным XML и XAML документом. В большинстве случаев корневым элементом будет являться один из

нескольких стандартных, таких как `<Window>`, `<Page>`, `<UserControl>`, `<ResourceDictionary>` или `<Application>`, в зависимости от предназначения файла.

Типичный XAML-файл выглядит примерно так:

XAML

```
<Window xmlns="http://schemas.microsoft.com/
          winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/
          winfx/2006/xaml">
</Window>
```

Как видно из примера, корневой элемент также содержит два атрибута: `xmlns` и `xmlns:x`. Эти атрибуты указывают анализатору XAML, какие пространства имен содержат описание типов данных, на которые ссылаются элементы. Атрибут `xmlns` описывает пространство имен по умолчанию. Элементы, описывающие объекты из этого пространства имен могут не содержать префикса в своем имени. `xmlns:x` определяет пространство имен для некоторых дополнительных конструкций языка XAML.

Стоит заметить, что объявление атрибутов, отвечающих за пространства имен достаточно поместить только в корневой элемент, и они будут видны во всех вложенных. Синтаксически возможно сделать подобное объявление для любого элемента документа, и опять же, такое объявление будет работать и для всех его дочерних элементов. Однако, такой подход довольно быстро захламит файл и понизит читабельность, поэтому принято объявлять пространства имен на уровне корневого элемента.

Если вы хотите использовать в разметке XAML тип данных, который объявлен в пространстве имен отличном от того, что используется по умолчанию, то необходимо будет объявить псевдоним для этого пространства имен в пределах XAML-файла.

В качестве примера давайте представим следующую ситуацию. Кто-то создал собственный элемент управления и поставляет его в виде отдельной сборки `Controls.dll`. В этой сборке элемент управления описан в классе `ProgressBars.RoundProgressBar`. Ниже показан фрагмент кода, который позволяет использовать этот элемент в вашем проекте. При этом, конечно же, сборка `Controls.dll` должна быть подключена к проекту.

XAML

```
<Window xmlns="http://schemas.microsoft.com/
          winfx/2006/xaml/presentation"
        xmlns:controls="clr-namespace:ProgressBars;
          assembly=Controls"
        xmlns:x="http://schemas.microsoft.com/
          winfx/2006/xaml">
    <controls:RoundProgressBar/>
</Window>
```

В данном случае, для пространства имен `ProgressBars` был выбран псевдоним `controls`. Описывая пространства имен в XAML вы вправе придумать любое название для псевдонима, которое вам больше всего подходит в данной ситуации. На самом деле название для пространства имен `x`, которое было объявлено выше, не более, чем соглашение. Вы можете и его назвать как-то по-другому.

Главное правило при использовании элементов из других пространств имен заключается том, что при указании их имени придется всегда указывать префикс.

В случае, если вы хотите создать псевдоним для пространства имен, описанного в той же самой сборке, в которой находится XAML-разметка, которая будет его использовать, при объявлении псевдонима можно опустить часть описывающую название сборки, в которой находится пространство имен.

XAML

```
<Window xmlns="http://schemas.microsoft.com/
          winfx/2006/xaml/presentation"
        xmlns:c="clr-namespace:WpfApplication.Controls"
        xmlns:x="http://schemas.microsoft.com/
          winfx/2006/xaml">
    <c:FancyButton/>
</Window>
```

5.2. Модель Code-Behind

Модель code-behind описывает каким образом XAML-разметка объединяется с программным кодом. XAML предоставляет средства для того, чтобы со стороны разметки можно было ассоциировать файл содержащий программный код с файлом разметки. Каким именно образом XAML и программный код должны интегрироваться вместе решает не XAML. Этим занимается программная платформа, использующая его (в данном случае WPF).

Фактически происходит следующее: описание одного класса разделяется на две части. Одна часть описывает фрагмент пользовательского интерфейса, который реали-

зуется данным классом, в виде XAML-разметки. Другая часть содержит программный код, описывающий логику поведения этого пользовательского интерфейса.

Из этого описания должно быть очевидным, что модель code-behind подразумевает использование двух файлов: файл с разметкой ([.xaml](#)) и файл с кодом (например, [.cs](#)).

При использовании модели code-behind необходимо соблюдать ряд ограничений и требований:

- Описание частичного класса должно быть наследником типа данных, который соответствует корневому элементу XAML-разметки. На самом деле, при описании частичного класса в файле с кодом можно даже не писать, что он наследуется от кого-то. Компилятор автоматически это допишет. Однако настоятельно не рекомендуется это делать, так как это вносит неясность в код и считается плохой практикой.
- Методы, являющиеся обработчиками событий, описанные в частичном классе должны быть экземплярными, и не могут быть статическими.
- Сигнатура метода, который является обработчиком события, должна полностью совпадать с делегатом соответствующего события.

Пример с детальным описанием модели code-behind будет рассмотрен далее.

Атрибуты **Name** и **x:Name**

При описании элементов в XAML-разметке можно использовать два их атрибута с очень похожими названиями: **Name** и **x:Name**. Атрибут **x:Name** является концепцией, принадлежащей XAML. Фактически данный

атрибут означает, что необходимо создать поле класса в файле с кодом модели code-behind. Атрибут **Name** является свойством объектов, принадлежащих WPF, которое хранит логическое имя элемента визуального дерева. Также WPF-атрибут **Name** является псевдонимом для XAML-атрибута **x:Name**, что делает выбор между ними не более чем стилем кодирования. Однако, стоит помнить, что концептуально данные атрибуты отличаются и принадлежат разным технологиям.

6. Обзор базовых классов элементов визуального дерева

В последующих разделах будут рассматриваться разнообразные элементы, которые могут присутствовать в визуальном дереве. Элементы, обладающие схожей функциональностью, наследуются от одних и тех же базовых классов, в которых эта функциональность реализуется. При этом часть иерархии классов (рис. 9) практически у всех этих классов идентична. Поэтому в данном разделе вкратце будут рассмотрены самые базовые классы и взаимосвязи между ними. По мере необходимости они будут рассматриваться детальнее.

Несмотря на то, что далеко не все классы на этой диаграмме являются абстрактными, создавать их экземпляры на прямую не стоит. В большинстве ситуаций вам будет необходимо создавать объекты производных от них классов.

6.1. Класс **DispatcherObject**

Практически каждый WPF-элемент имеет, так называемую, привязку к потоку. Другими словами, доступ к такому элементу должен осуществляться только с того потока, который его создал. Если попытаться получить доступ к свойствам элемента на чтение или запись из потока, в котором он не создавался, то выбросится исключение типа `System.InvalidOperationException`, в сообщении которого будет описана соответствующая ошибка.

6. Обзор базовых классов элементов визуального дерева

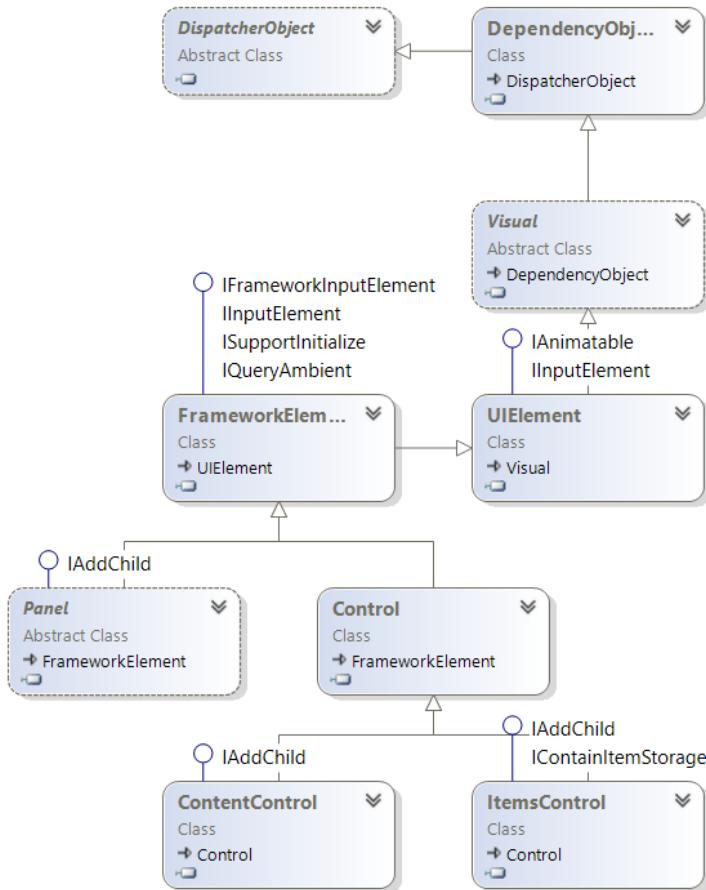


Рис. 9. Базовые классы элементов визуального дерева

Для того, чтобы реализовать такую функциональность, каждый объект требующий привязки к потоку в конечном счете наследуется от класса `System.Windows.Threading.DispatcherObject`. Этот класс предоставляет свойство, которое возвращает объект, при помощи которого выполняются необходимые операции над элементом, с которым он ассоциирован. Этот объект-диспетчер обладает

очередью задач, и отвечает за то, чтобы они выполнялись в потоке элемента.

Если проект не использует многопоточность при работе с элементами интерфейса, то описанная выше ситуация не страшна.

6.2. Класс DependencyObject

По умолчанию, любое свойство, созданное в .NET Framework, имеет очень базовую функциональность. Класс System.Windows.DependencyObject используется для реализации системы свойств WPF, которая немного отличается от привычной, предоставляя расширенный набор возможностей, таких как: оповещение при изменении значения, валидация, привязка к стилям, значение по умолчанию и др.

6.3. Класс Visual

Класс System.Windows.Media.Visual предоставляет набор методов и свойств, которые необходимы для отображения элементов на экране. Также класс предоставляет функциональность для взаимодействия с визуальным деревом и проверки на визуальное попадание курсора на элемент (*hit test*).

6.4. Класс UIElement

Класс System.Windows.UIElement содержит большое количество методов, свойств и событий, которые, в основном, отвечают за взаимодействие с пользователем. Среди реализованной функциональности можно отметить: получение данных посредством клавиатуры, мыши и других устройств ввода, управление фокусом ввода,

а также управление общим статусом элемента (видимость, включенность и т.д.).

6.5. Класс FrameworkElement

Класс System.Windows.FrameworkElement добавляет еще больше функциональности в WPF-элементы, которая необходима для реализации системы компоновки элементов, логического дерева, стилей и привязки данных. Также здесь содержится набор событий, описывающих жизненный цикл элементов. При помощи них можно определять, например, когда элемент инициализируется, загружается или выгружается.

6.6. Класс Panel

Класс System.Windows.Controls.Panel является базовым для панелей, которые отвечают за упорядочивание и расположение помещенных в них элементов. Панели являются основным механизмом компоновки пользовательских интерфейсов в WPF.

6.7. Класс Control

Класс System.Windows.Controls.Control является базовым классом для всех элементов управления, главной отличительной чертой которых является наличие шаблона, отвечающего за их визуализацию.

6.8. Класс ContentControl

Класс System.Windows.Controls.ContentControl является базовым для всех элементов управления, которые содержат только один дочерний элемент.

6.9. Класс ItemsControl

Класс System.Windows.Controls.ItemsControl является базовым для всех элементов управления, которые могут содержать несколько дочерних элементов.

7. Обзор WPF-приложения

В данном разделе будет подробно рассмотрен процесс создания WPF-приложения, его структура, а также использование модели code-behind.

7.1. Создание WPF-проекта

Процесс создания WPF-приложения в среде разработки Visual Studio практически ничем не отличается от процесса создания любого другого проекта .NET Framework. В первую очередь необходимо выбрать пункт меню **File → New → Project...**, или нажать комбинацию клавиш **Ctrl+Shift+N** (рис. 10).

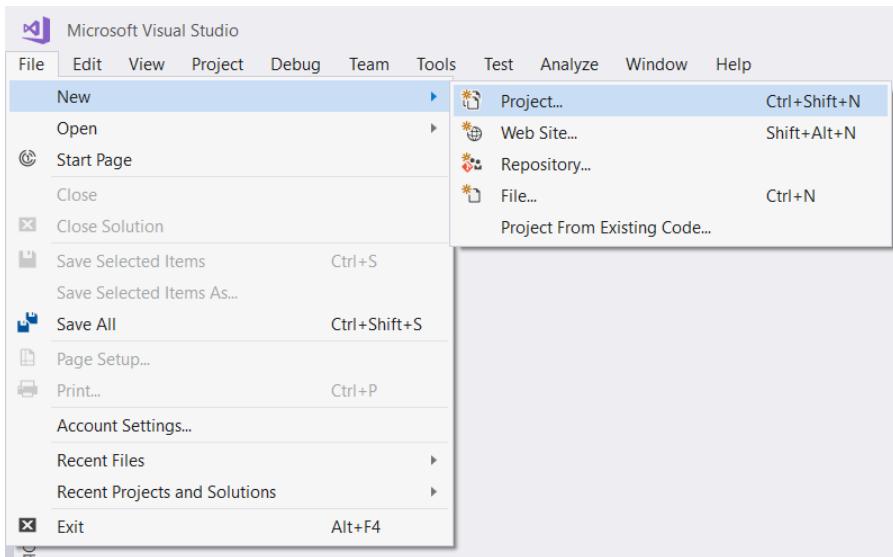


Рис. 10. Выбор пункта главного меню
для создания нового проекта

В открывшемся окне, слева, необходимо выбрать **Templates → Visual C# → Windows Classic Desktop**, после чего выбрать тип проекта **WPF App (.NET Framework)**. В нижней части окна, как и в других типах проектов, находятся поля **Name** и **Location**, которые вы можете изменить, если хотите поменять имя проекта или его месторасположение соответственно (рис. 11).

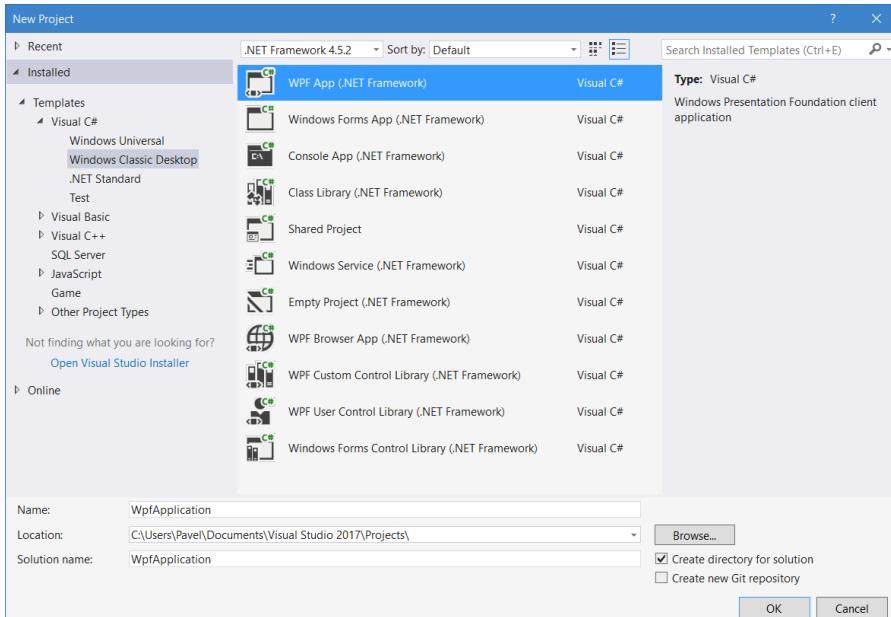


Рис. 11. Выбор типа проекта

7.2. Структура WPF-приложения

Каждый новый проект обладает одинаковым содержимым, так называемой «заглушки». Вы можете сразу же скомпилировать проект и увидеть готовое приложение, состоящие всего лишь из пустого окна. В разных версиях Visual Studio, автоматически сгенерированный

начальный код и набор подключенных по умолчанию сборок может отличаться. К тому же, часть сгенерированного кода нужна не всегда, поэтому здесь мы будем рассматривать минимум кода, необходимого на данном этапе. Следовательно, если вы создадите проект и заметите в нем фрагмент кода, который не рассмотрен в этом разделе, с большой вероятностью вы можете его просто удалить. Начальная структура файлов нового проекта показана на рис. 12.

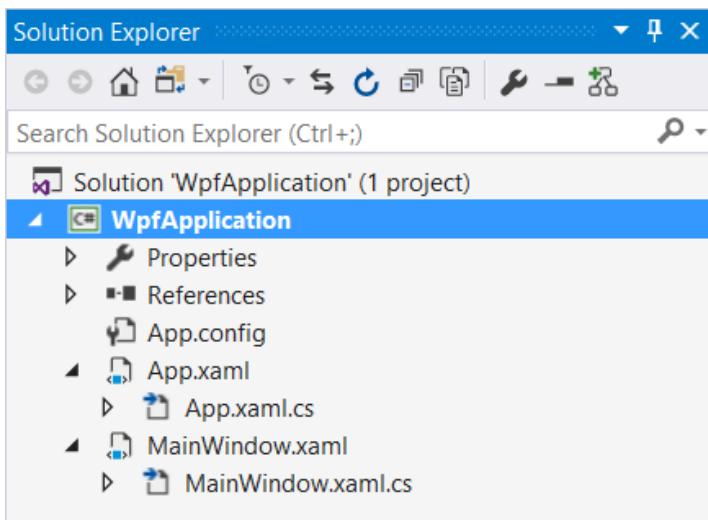


Рис. 12. Начальная структура файлов нового проекта

Здесь можно заметить, что файлы сгруппированы попарно. На самом деле, в папке с проектом находится 4 файла о которых будет идти речь в данном разделе: [App.xaml](#), [App.xaml.cs](#), [MainWindow.xaml](#), [MainWindow.xaml.cs](#). Из названий видно, что пары файлов описывают две отдельные сущности. Такая структура файлов необходимо для реализации модели code-behind.

Класс App

В WPF-приложении нет привычной точки входа, которой обычно является метод Main. Технически метод Main, конечно же, есть, так как это часть инфраструктуры .NET Framework, но разработчик не должен описывать его явным образом, потому что он сгенерируется автоматически. Вместо этого разработчик приложения должен создать класс, который будет наследоваться от класса System.Windows.Application. В нашем случае это класс App, описание которого находится в двух файлах: App.xaml и App.xaml.cs.

Компилятор генерирует метод Main примерно следующего вида:

C#

```
public static void Main()
{
    var application = new App();
    application.InitializeComponent();
    application.Run();
}
```

Метод App.InitializeComponent также генерируется автоматически и необходим для того, чтобы настроить главное окно приложения, а метод System.Windows.Application.Run запускает приложение.

Файл App.xaml может выглядеть следующим образом:

XAML

```
<Application x:Class="WpfApplication.App"
    xmlns="http://schemas.microsoft.com/
    winfx/2006/xaml/presentation"
```

```

xmlns:x="http://schemas.microsoft.com/
winfx/2006/xaml"
StartupUri="MainWindow.xaml"/>

```

Самой интересной частью этой разметки является атрибут **x:Class="WpfApplication.App"**. Это ключевой момент при реализации модели code-behind. Именно таким образом указывается в каком месте содержится программный код, относящийся к этому файлу разметки. В качестве значения атрибута **x:Class** должно быть задано название соответствующего класса с указанием полного пространства имен.

Другим не маловажным моментом здесь является атрибут **StartupUri**, который содержит название файла разметки, содержащего описание главного окна приложения. Путь к файлу здесь записывается относительный, который вычисляется относительно корневой директории проекта. Если бы структура директорий была бы такой, как показано на рис. 13, то значение атрибута **StartupUri** было бы **Windows/MainWindow.xaml**.

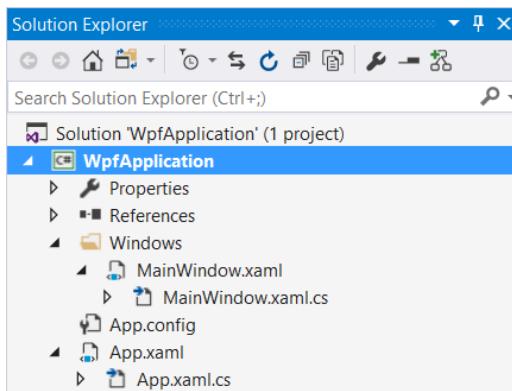


Рис. 13. Альтернативная структура файлов проекта

Файл App.xaml.cs содержит следующий код:

C#

```
namespace WpfApplication
{
    public partial class App : Application
    {
    }
}
```

Как видите, в данном файле описан всего лишь пустой класс [App](#). Самым важным здесь является указание того, что он наследуется от класса [System.Windows.Application](#), и является частичным ([partial](#)). Данный класс, конечно же, может содержать определенный код, но это не обязательно.

Также очень важным является то, чтобы класс обладал конструктором по умолчанию (который может быть объявлен с использованием любого спецификатора доступа). По умолчанию все классы, которые используют модель code-behind должны быть помечены спецификатором доступа [public](#). По очевидным причинам, это может быть не всегда удобно. В определенных ситуациях может потребоваться, чтобы класс обладал спецификатором доступа [internal](#). Чтобы это сработало необходимо будет добавить в XAML-файл, в корневой элемент, который содержит атрибут [x:Class](#), еще один атрибут — [x:ClassModifier](#). Значение этого атрибута отличается в зависимости от того, какой язык программирования используется в связке с XAML. Для C# значение этого атрибута должно быть [internal](#), т.е. разметка будет выглядеть так:

XAML

```
<Application x:Class="WpfApplication.App"
             x:ClassModifier="internal"
             xmlns="http://schemas.microsoft.com/
winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/
winfx/2006/xaml"
             StartupUri="MainWindow.xaml"/>
```

Стоит упомянуть о необходимости согласованности этих спецификаторов доступа. Если вы указали в разметке атрибут **x:ClassModifier** со значением **internal**, то класс в коде обязан быть помечен таким же спецификатором доступа. То же самое касается и согласованности частичных описаний публичного класса.

Класс Application

- Статические свойства класса **System.Windows.Application**:
- **Current** (**System.Windows.Application**). Получает объект текущего приложения.
- Свойства класса **System.Windows.Application**:
- **MainWindow** (**System.Windows.Window**). Получает или задает главное окно приложения.
- **ShutdownMode** (**System.Windows.ShutdownMode**). Получает или задает режим закрытия приложения. Значение по умолчанию — **System.Windows.ShutdownMode.OnLastWindowClose**.
- **StartupUri** (**System.Uri**). Получает или задает URI, который описывает, что именно должно быть показано при запуске приложения.

- **Windows** (System.Windows.WindowCollection). Получает коллекцию созданных окон в приложении.
- События класса System.Windows.Application:
- **Activated** (System.EventHandler). Срабатывает, когда приложение становится активным.
- **Deactivated** (System.EventHandler). Срабатывает, когда приложение перестает быть активным.
- **Exit** (System.Windows.ExitEventHandler). Срабатывает непосредственно перед тем, как приложение будет закрыто. Данное событие не может быть отменено.
- **Startup** (System.Windows.StartupEventHandler). Срабатывает, когда вызывается метод System.Windows.Application.Run. Фактически, это самое раннее место в жизненном цикле WPF-приложения, доступ к которому можно получить. Обычно здесь производится настройка ресурсов и свойств уровня всего приложения.
- Методы класса System.Windows.Application:
- **Run()**. Запускает приложение.
- **Run(window)**. Запускает приложение и отображает указанное окно.
- **Shutdown()**. Завершает приложение и возвращает операционной системе код завершения — 0.
- **Shutdown(exitCode)**. Завершает приложение и возвращает операционной системе указанный код завершения.

Для того, чтобы указать режим завершения приложения необходимо использовать перечисление System.Windows.ShutdownMode, которое содержит следующие варианты:

- OnExplicitShutdown. Приложение будет закрыто только при явном вызове метода System.Windows.Application.Shutdown.
- OnLastWindowClose. Приложение будет закрыто, если будет закрыто последнее окно приложения или будет явно вызван метод System.Windows.Application.Shutdown.
- OnMainWindowClose. Приложение будет закрыто, если будет закрыто главное окно приложения или будет явно вызван метод System.Windows.Application.Shutdown.

События в стандартных классах WPF реализованы, согласно рекомендаций Microsoft. Это означает, что помимо события, в классе также присутствует метод, который можно переопределить.

Ниже показан пример того, как можно подписаться на событие System.Windows.Application.Startup.

C#

```
public partial class App : Application
{
    public App()
    {
        Startup += App_Startup;
    }

    private void App_Startup(object sender,
                           StartupEventArgs e)
    {
        // Startup code...
    }
}
```

Другой вариант — переопределение метода:

C#

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        // Startup code...
    }
}
```

Показанные выше примеры полностью аналогичны по своей функциональности. Однако стоит обратить внимание на строку `base.OnStartup(e)`. Если ее убрать, то логика работы изменится. Вызов базовой версии метода `System.Windows.Application.OnStartup` необходим для того, чтобы оповестить всех подписчиков события `System.Windows.Application.Startup`. Другими словами, если скомбинировать оба варианта вместе и убрать указанную строку, то метод `App.App_Startup` не будет вызван:

C#

```
public partial class App : Application
{
    public App()
    {
        Startup += App_Startup;
    }

    private void App_Startup(object sender,
                            StartupEventArgs e)
    {
```

```

        // Startup code that would never be executed...
    }

protected override void OnStartup(StartupEventArgs e)
{
    // Startup code...
}
}

```

Все, что было описано касательно события System.Windows.Application.Startup применимо практически ко всем событиям в WPF-классах, которые будут рассматриваться далее.

Класс MainWindow

Другая группа файлов (MainWindow.xaml и MainWindow.xaml.cs) содержит описание класса главного окна приложения. Эта группа файлов также использует модель code-behind, и к ней применимы все правила, касающиеся ее, которые были описаны для класса App.

Файл MainWindow.xaml устроен примерно следующим образом:

XAML

```

<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/
        winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/
        winfx/2006/xaml"
        Height="350"
        Title="MainWindow"
        Width="525">
    <Grid></Grid>
</Window>

```

Элемент `<Window>` описывает окно и, наверное, является наиболее часто используемым корневым элементом в WPF. Атрибут `x:Class`, по-прежнему, содержит название частичного класса, в котором находится программный код модели code-behind. Атрибуты `Width` и `Height` задают размеры окна, а атрибут `Title` — заголовок.

Элемент `<Window>` может содержать только один дочерний элемент, поэтому в качестве варианта по умолчанию сюда помещена панель сетка (`<Grid>`), которая является самой «продвинутой» из всех. Вы можете изменить ее на любой другой элемент управления или другую панель.

Файл `MainWindow.xaml.cs` содержит следующий код:

C#

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

Здесь самым важным является вызов метода `MainWindow.InitializeComponent`, который генерируется автоматически. Именно в этом методе происходит анализ соответствующего XAML-файла и создание объектов по его описанию.

Класс `Window`

На рис. 14 показан класс `System.Windows.Window` и его базовый класс.

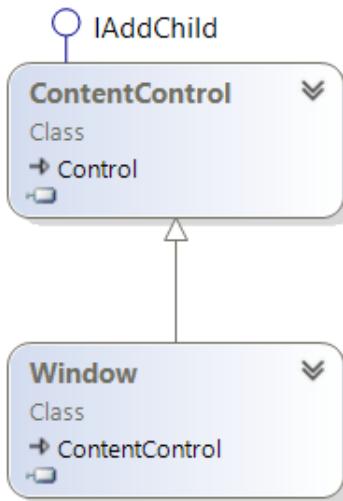


Рис. 14. Диаграмма классов, описывающих окно

Свойства класса System.Windows.Window:

- AllowsTransparency (System.Boolean). Получает или задает значение, которое указывает поддерживает ли клиентская область окна прозрачность. **true** если клиентская область поддерживает прозрачность; иначе — **false**. Значение по умолчанию — **false**.
- DialogResult (System.Nullable<System.Boolean>). Получает или задает значение, которое будет возвращено из метода System.Windows.Window.ShowDialog. Значение по умолчанию — **false**.
- Icon (System.Windows.Media.ImageSource). Получает или задает иконку окна.
- IsActive (System.Boolean). Получает значение, которое указывает является ли окно активным. **true** если окно является активным; иначе — **false**.

- **Left** (System.Double). Получает или задает позицию левой границы окна относительно левой границы рабочего стола.
- **OwnedWindows** (System.Windows.WindowCollection). Получает коллекцию окон, для которых текущее окно является владельцем.
- **Owner** (System.Windows.Window). Получает или задает окно, которое является владельцем текущего окна.
- **ResizeMode** (System.Windows.ResizeMode). Получает или задает режим изменения размеров окна. Значение по умолчанию — System.Windows.ResizeMode.CanResize.
- **RestoreBounds** (System.Windows.Rect). Получает размер окна, до того, как оно было свернуто или развернуто на весь экран.
- **ShowActivated** (System.Boolean). Получает или задает значение, которое указывает должно ли окно быть активировано при первом показе. **true** если окно должно быть активировано при первом показе; иначе — **false**. Значение по умолчанию — **true**.
- **ShowInTaskbar** (System.Boolean). Получает или задает значение, которое указывает должно ли окно отображаться на панели задач. **true** если окно должно отображаться на панели задач; иначе — **false**. Значение по умолчанию — **true**.
- **SizeToContent** (System.Windows.SizeType). Получает или задает режим подгонки размеров окна под содержимое. Значение по умолчанию — System.Windows.SizeType.Manual.

- **Title** (System.String). Получает или задает заголовок окна.
- **Top** (System.Double). Получает или задает позицию верхней границы окна относительно верхней границы рабочего стола.
- **Topmost** (System.Boolean). Получает или задает значение, которое указывает должно ли окно находиться поверх остальных окон. **true** если окно должно находиться поверх остальных окон; иначе — **false**. Значение по умолчанию — **false**.
- **WindowStartupLocation** (System.Windows.WindowStartupLocation). Получает или задает начальную позицию окна при первом показе. Значение по умолчанию — System.Windows.WindowStartupLocation.Manual.
- **WindowState** (System.Windows.WindowState). Получает или задает состояние окна (свернутое, развернутое, восстановленное). Значение по умолчанию — System.Windows.WindowState.Normal.
- **WindowStyle** (System.Windows.WindowStyle). Получает или задает стиль рамки окна. Значение по умолчанию — System.Windows.WindowStyle.SingleBorderWindow.
- События класса System.Windows.Window:
- **Activated** (System.EventHandler). Срабатывает, при активации окна (переносится на передний план).
- **Closed** (System.EventHandler). Срабатывает, когда окно закрывается.
- **Closing** (System.ComponentModel.CancelEventHandler). Срабатывает непосредственно после вызова метода System.Windows.Window.Close, и перед событием

System.Windows.Window.Closed. Может быть обработано, чтобы отменить процесс закрытия окна.

- **ContentRendered** (System.EventHandler). Срабатывает, когда содержимое окна отображено.
- **Deactivated** (System.EventHandler). Срабатывает, когда окно перестает быть активным (переносится на задний план).
- **LocationChanged** (System.EventHandler). Срабатывает, когда изменяется позиция окна.
- **SourceInitialized** (System.EventHandler). Срабатывает, когда инициализируется дескриптор окна.
- **StateChanged** (System.EventHandler). Срабатывает, когда изменяется значение свойства System.Windows.Window.WindowState.
- Методы класса System.Windows.Window:
 - **Activate()**. Переносит окно на передний план. Возвращает **true**, если операция выполнилась успешно; иначе — **false**.
 - **Close()**. Закрывает окно.
 - **Hide()**. Прячет окно.
 - **Show()**. Показывает окно.
 - **ShowDialog()**. Показывает окно и ожидает его закрытия. Возвращает значение, которое описывает результат взаимодействия с окном: действие было принято (**true**) или отменено (**false**).

Для того, чтобы указать режим изменения размеров окна необходимо использовать перечисление System.Windows.ResizeMode, которое содержит следующие варианты:

- NoResize. Размеры окна не могут быть изменены. Кнопки «свернуть» и «развернуть» не отображаются в заголовке окна.
- CanMinimize. Окно может быть свернуто и восстановлено. Кнопки «свернуть» и «развернуть» отображаются, но только кнопка «свернуть» активна.
- CanResize. Размеры окна могут быть изменены. Кнопки «свернуть» и «развернуть» отображаются и активны.
- CanResizeWithGrip. Размеры окна могут быть изменены. Кнопки «свернуть» и «развернуть» отображаются и активны. В нижнем правом углу окна отображается элемент захвата для изменения размеров окна (size grip).

Для того, чтобы указать режим подгонки окна под содержимое необходимо использовать перечисление System.Windows.[SizeToContent](#), которое содержит следующие варианты:

- Height. Высота окна будет автоматически подстраиваться под высоту содержимого, а ширина — нет.
- Manual. Размеры окна не будут автоматически подстраиваться под содержимое. Вместо этого размеры окна будут регулироваться набором свойств, отвечающих за явное указание размеров и допустимые диапазоны изменения размеров.
- Width. Ширина окна будет автоматически подстраиваться под ширину содержимого, а высота — нет.
- WidthAndHeight. Размеры окна будут автоматически подстраиваться под размеры содержимого.

Чтобы указать начальную позицию окна при первом показе необходимо использовать перечисление `System.Windows.WindowStartupLocation`, которое содержит следующие варианты:

- `CenterOwner`. Окно позиционируется по центру своего владельца, который описывается свойством `System.Windows.Window.Owner`.
- `CenterScreen`. Окно позиционируется по центру экрана.
- `Manual`. Начальная позиция окна определяется значениями, установленными явным образом или используется позиция по умолчанию.

Для того, чтобы указать состояние окна (свернутое, развернутое, восстановленное), необходимо использовать перечисление `System.Windows.WindowState`, которое содержит следующие варианты:

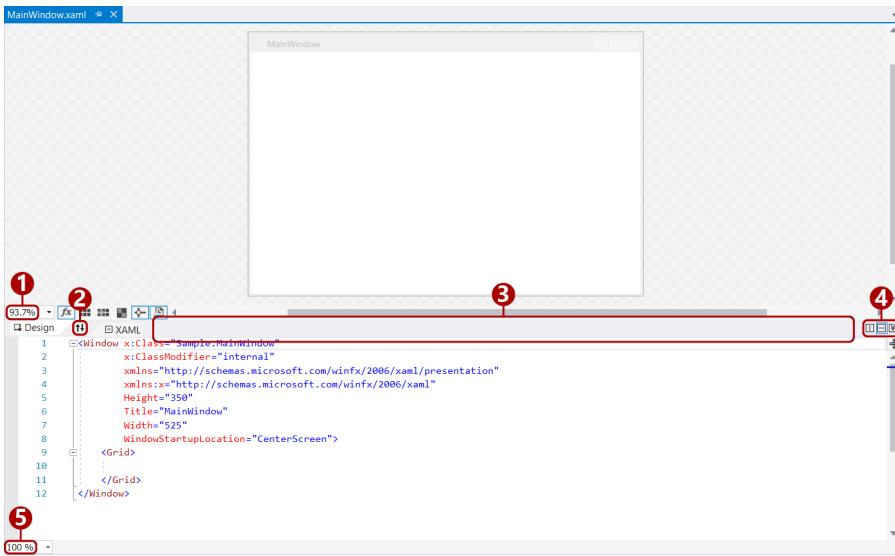
- `Maximized`. Окно развернуто.
- `Minimized`. Окно свернуто.
- `Normal`. Окно восстановлено.

Для того, чтобы указать стиль рамки окна необходимо использовать перечисление `System.Windows.WindowStyle`, которое содержит следующие варианты:

- `None`. Заголовок окна и рамка не отображается. Отображается только клиентская область окна.
- `SingleBorderWindow`. Окно с обычной рамкой.
- `ThreeDBorderWindow`. Окно с 3D-рамкой.
- `ToolWindow`. Окно с тонкой рамкой без кнопок «свернуть» и «развернуть».

8. Обзор редактора XAML

Открывая XAML-файл в Visual Studio, вы попадете в окно редактора разметки, показанное на рис. 15. Рабочая область этого окна разделена на две области. В верхней показывается, в реальном времени, результат текущей разметки, которая, в свою очередь, описывается в нижней области. При необходимости можно изменить размер



1. Редактируемый выпадающий список, позволяющий изменять масштаб верхней рабочей области.
2. Кнопка, позволяющая поменять местами рабочие области.
3. Ползунок, позволяющий изменять размеры каждой рабочей области за счет другой.
4. Кнопки переключения между горизонтальным и вертикальным разделением областей.
5. Редактируемый выпадающий список, позволяющий изменять масштаб нижней рабочей области.

Рис. 15. Окно редактора XAML

каждой из этих областей, поменять их местами, изменить масштаб или изменить разделение с горизонтального на вертикальное. Если вы будете взаимодействовать с любой из этих областей, внося изменения (редактировать разметку или перемещать элементы управления), то эти изменения сразу же будут синхронизированы и отображены в другой.

У вас, как у разработчика, есть несколько способов редактировать XAML-разметку, пользуясь данным редактором.

Самый очевидный способ — это редактирование разметки вручную. Для этого вам необходимо взаимодействовать с нижней частью редактора. Здесь нет никаких

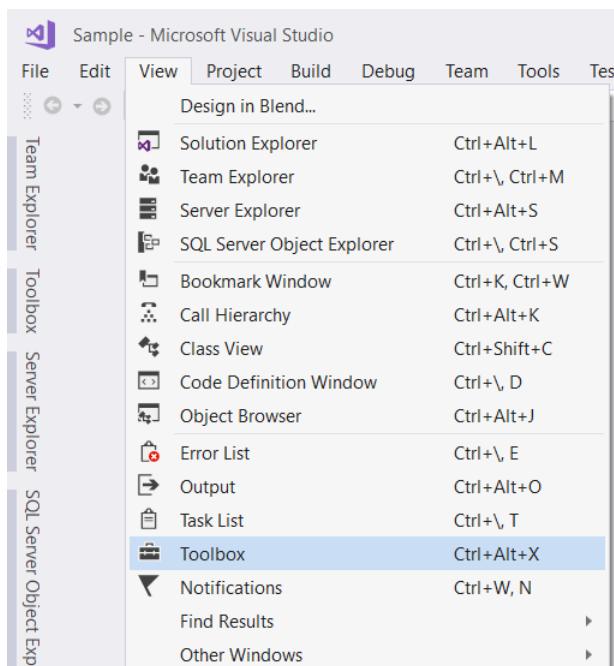


Рис. 16. Выбор пункта главного меню для открытия окна Toolbox

особенностей, о которых стоило бы говорить. Типичный текстовый редактор, с которым знаком любой разработчик, который ранее взаимодействовал с Visual Studio.

Следующий способ подразумевает «визуальное» редактирование разметки. Для такого вида редактирования понадобится взаимодействовать с дополнительными окнами Visual Studio. Первое из них — Toolbox. Для того, чтобы открыть это окно необходимо выбрать пункт меню **View → Toolbox**, или нажать комбинацию горячих клавиш **Ctrl+Alt+X** (рис. 16).

Окно Toolbox (рис. 17) содержит список элементов (элементы управления, примитивные элементы, панели), которые можно использовать для создания разметки.

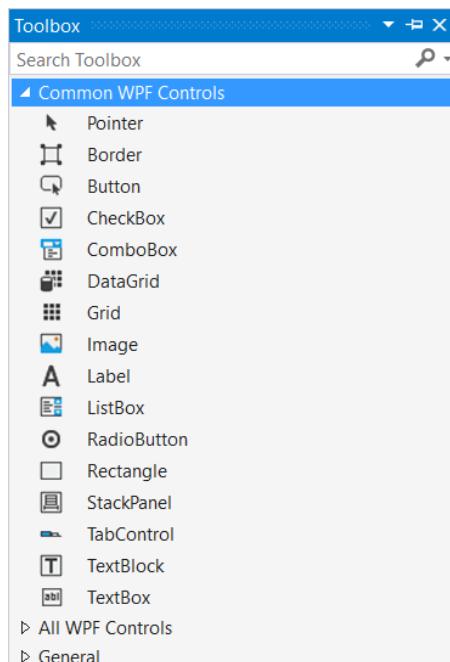


Рис. 17. Окно Toolbox

Некоторые окна в Visual Studio могут работать по-разному, в зависимости от того, какое окно или элемент интерфейса является активным в данный момент. Toolbox работает именно по такому принципу. Для того, чтобы это окно было заполнено элементами, которые можно использовать для создания интерфейса, необходимо, чтобы окно редактирования XAML-разметки было активно. Если у вас будет активно какое-то другое окно, например, редактор C#-кода, то Toolbox будет иметь другой вид.

Если взглянуть на это окно, то можно увидеть два раскрывающихся пункта: Common WPF Controls и All WPF Controls. Они содержат наиболее часто используемые элементы для WPF разработки и все доступные элементы соответственно.

Для того, чтобы использовать какой-то из них, необходимо навести курсор мыши на требуемый элемент, нажать и удерживать левую кнопку мыши и перетащить его на область с окном (верхняя часть редактора XAML) и отпустить кнопку. Таким же образом можно перемещать уже существующие элементы. При этом, в XAML файл автоматически допишется фрагмент разметки, который будет отображать выполненные манипуляции.

Другим вариантом помещения элемента в пределы визуальной части редактора является двойной щелчок левой кнопки мыши по требуемому элементу в Toolbox. В таком случае, он поместить как дочерний элемент для активного (выбранного) в пределах разметки.

Для настройки созданных элементов необходимо другое окно — **Properties**. Для того чтобы открыть это

окно, необходимо выбрать пункт меню **View → Properties Window**, или нажать **F4** (рис. 18).

Окно **Properties** (рис. 19) позволяет настраивать свойства и события элементов разметки. В этом окне всегда показывается информация об активном (выбранном) элементе разметки. Устроено данное окно крайне просто. Свойства каждого элемента поделены на категории (цвет, расположение, текст и др.) и отображаются в виде двух

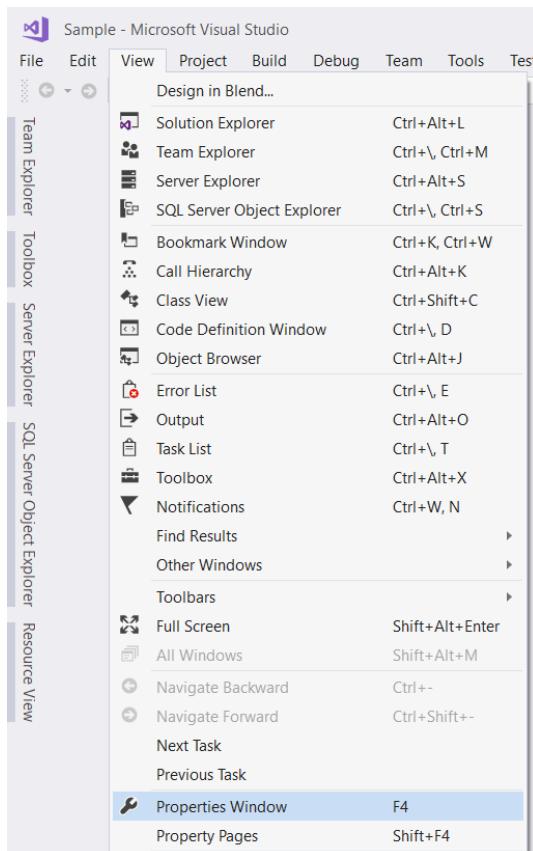
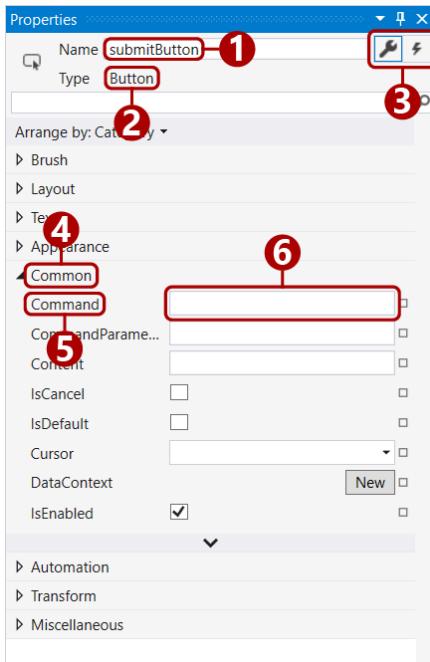


Рис. 18. Выбор пункта главного меню для открытия окна Properties



1. Поле для ввода текста, содержащее имя элемента управления, необходимое для взаимодействия с ним из кода.
2. Метка, отображающая тип данных элемента управления.
3. Кнопки для переключения окна Properties в режим отображения свойств или событий.
4. Метка, отображающая название категории свойств.
5. Метка, отображающая название свойства.
6. Элемент управления, позволяющий редактировать значение свойства.

Рис. 19. Окно Properties

колонок: название свойства и элемент управления для задания значения свойства.

Редактор XAML и окно Properties полностью синхронизированы между собой. Имеется ввиду, что, изменив разметку посредством редактора XAML каким-то образом, соответствующие значения в окне Properties тоже изменятся, и наоборот.

9. Компоновка

При создании пользовательского интерфейса приложения очень много усилий уходит на то, чтобы сделать его привлекательным и удобным для использования. Но не меньше усилий требует и другой его аспект — гибкость. Под гибкостью интерфейса следует понимать его способность справляться с изменениями размеров окна и его наполнением. При изменении размеров окна или его части, интерфейс должен адаптироваться под новые размеры адекватным образом, при необходимости увеличивая или уменьшая размеры определенных его частей.

Другой, не менее распространенной, причиной, требующей изменений в интерфейсе, является изменение содержимого элементов управления. К примеру, локализованные приложения поддерживают несколько вариантов перевода своего интерфейса на другие языки. Слова и фразы, из которых состоит интерфейс, в разных языках имеют разную длину, а, следовательно, и размеры, что является проблемой при использовании фиксированных размеров и позиционирования элементов управления. Это является проблемой, потому что интерфейс, спроектированный с использованием элементов управления, которые идеально вмещают строки одного языка, будет выглядеть совершенно иначе, при использовании другого языка: некоторые элементы будут урезать свое содержимое, показывая только часть строки, а другие будут содержать слишком много неиспользуемого пространства. Все эти

проблемы связаны с компоновкой, т.е. с определением расположения элементов и их размеров.

При использовании технологий, которые предшествовали появлению WPF, возможности по управлению компоновкой интерфейса были, мягко говоря, не очень обширны. В основном, они сводились к заданию фиксированного размера и местоположения каждого элемента управления. Если же приложение должно было обладать возможностью изменения размера окна, то разработчику приходилось писать код, изменяющий размеры и расположение элементов управления вручную. Некоторые технологии позволили автоматизировать самые базовые сценарии поведения интерфейса при изменении размеров. Например, в Windows Forms, есть понятия закрепления и пристыковки элемента к сторонам окна. Однако, это помогает только в случаях с очень простым интерфейсом. В более сложных ситуациях приходилось разрабатывать собственный алгоритм, описывающий поведение элементов управления.

WPF решает описанные выше проблемы путем изменения процесса компоновки интерфейса. Вместо того, чтобы задавать каждому элементу интерфейса фиксированные координаты и размер, предлагается использовать набор панелей. Каждая панель обладает своей собственной логикой упорядочивания помещенных в нее элементов.

В основе модели компоновки, заложенной в WPF, лежат два основных правила:

- Элементам управления не стоит задавать фиксированных размеров. Вместо этого, их размеры должны определяться их содержимым и логикой панели,

в которой они находятся. При этом, элементы можно ограничивать, задавая минимальные и максимальные размеры, что позволит им адаптироваться под изменения, но только в заданном диапазоне.

- Элементы управления не должны обладать жестко заданными координатами. Они должны упорядочиваться при помощи логики панелей, в которых они находятся и заданных им отступов.

Как из любого правила, из этих правил также бывают исключения. Встречаются ситуации, в которых необходимо задать фиксированный размер или местоположение элемента. Вы должны стараться избегать их если это возможно и расценивать их как крайнюю меру, так как это противоречит философии компоновки WPF.

9.1. Этапы компоновки

Перед тем как перейти к рассмотрению отличительных особенностей каждой из существующих панелей, стоит рассмотреть, по какому общему принципу все они работают.

Расположение элементов вычисляется в нескольких ситуациях: при первом показе и, если происходит что-то, что влияет на размеры или расположение (например, изменение содержимого или размеров окна). Компоновка происходит в два этапа: этап измерений и этап расстановки.

На этапе измерений осуществляется обход всех элементов визуального дерева и получение предпочтительных размеров для каждого из них. Для этого вызывается метод `System.Windows.UIElement.Measure`, в который передается, когда это возможно, доступный размер, который

может повлиять на предпочтительный размер элемента. Например, если у элемента, отображаемого текст, включен перенос по словам, то такой элемент будет требовать больше высоту при уменьшении ширины доступного пространства. Однако, не всегда будет известно, сколько пространства доступно. Элементы, содержащие области с прокруткой предоставляют бесконечное «виртуальное» пространство, т.е. они могут содержать элементы любого размеры, при этом показывая только часть их. В таких случаях, в качестве доступного пространства будет передана величина, описывающая бесконечно доступное пространство. Исходя из этого, компоновка бывает двух видов: ограниченная и неограниченная.

Ограниченнaя компоновка происходит в тех случаях, когда известно, сколько есть доступного пространства, которое необходимо разделить между элементами. Неограниченная — в ситуациях, когда размеры доступного пространства неизвестны или не ограничены. Также стоит заметить, что при компоновке есть разграничение между вертикальным и горизонтальным пространством. В качестве примера давайте представим, что доступна область только с вертикальной полосой прокрутки. Это означает, что она обладает неограниченным вертикальным пространством и ограниченным горизонтальным. В таком случае в метод `System.Windows.UIElement.Measure` будет передано определенное число описывающее ширину доступной области и положительная бесконечность (`System.Double.PositiveInfinity`) в качестве высоты.

При обходе, элементы управления не могут потребовать неограниченного пространства. Они обязаны вернуть

конечный предпочтительный размер, не зависимо от того, сколько пространства доступно. В случае неограниченной компоновки, элементы возвращают минимально необходимый им размер. Это еще называется размер в соответствии с содержимым (*size to content*). На самом деле, неограниченная компоновка применяется гораздо чаще, чем может показаться на первый взгляд. Такой вид компоновки применяется каждый раз, когда необходимо узнать «реальные» размеры элемента. Это очень полезно даже в таких простых операциях, как выравнивание по правому краю или центрирование. В этих ситуациях, необходимо знать размер элемента, чтобы добавить необходимые отступы. Поэтому не зависимо от интерфейса, часто применяется смесь ограниченной и неограниченной компоновки для вычисления расположения элементов.

После завершения первого этапа становится известно, какие размеры необходимы каждому элементу визуального дерева, и осуществляется второй проход по нему. В этот раз вызывается метод `System.Windows.UIElement.Arrange` для каждого элемента, в который передается вычисленный размер и расположение элемента. Очевидно, что не все элементы всегда смогут получить предпочтительный размер. В этих случаях могут происходить разные вещи. Некоторые элементы могут быть обрезаны, а некоторые могут вполне нормально отображаться даже при меньшем размере. Например, элемент отображаемый изображение, в качестве предпочтительного размера потребует размер самого изображения. При этом, получив меньший размер в качестве вычисленного, изображение может быть просто сжато и показано в меньшем варианте.

Такая двухэтапная компоновка для разработчиков выглядит следующим образом. Вместо того, чтобы самим вычислять размеры и расположение элементов, разработчик описывает требования к компоновке в декларативной форме в виде XAML-разметки, тем самым перекладывая работу по вычислениям точных значений на плечи WPF.

10. Аппаратно-независимые единицы

В WPF часто размерности задаются в так называемых аппаратно-независимых единицах, размер которых равен 1/96 дюйма. Используются именно такие единицы, вместо реальных пикселей для того, чтобы получить одинаковые размеры элементов пользовательского интерфейса на любом экране. Фактически, это означает, что если вы зададите размер элемента равный 96 таким единицам, то он должен равняться 1 дюйму независимо от того, сколько пикселей помещается в дюйм в используемом мониторе.

11. Панели

Все панели в WPF наследуются от базового класса `System.Windows.Controls.Panel` (рис. 20).

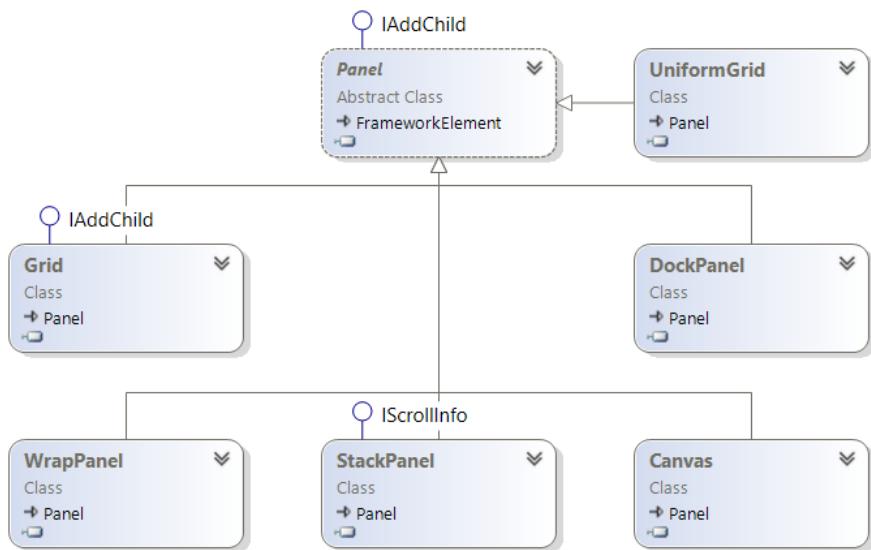


Рис. 20. Диаграмма классов, описывающих панели

Свойства класса `System.Windows.Controls.Panel`:

- **Background** (`System.Windows.Media.Brush`). Получает или задает кисть, которая используется для заливки области между границами панели. Значение по умолчанию — `null`.
- **Children** (`System.Windows.Controls.UIElementCollection`). Получает коллекцию элементов, помещенных в панель. В ней хранятся дочерние элементы только первого уровня. Значение по умолчанию — пустая коллекция.

- **IsItemsHost** (System.Boolean). Получает или задает значение, которое указывает, является ли панель контейнером для дочерних элементов, сгенерированных элементами управления, которые наследуются от класса System.Windows.Controls.ItemsControl. true если является; иначе — false. Значение по умолчанию — false.

Присоединенные свойства класса System.Windows.Controls.Panel:

- **ZIndex** (System.Int32). Получает или задает значение z-порядка для элемента в панели. Чем больше значение, тем больше вероятность появления элемента на переднем плане. Например, элемент со значением z-порядка 5, будет показан поверх элемента со значением 4. Данное свойство может также принимать отрицательные значения. В случае, когда два элемента обладают одинаковым значением z-порядка, они будут показаны в порядке появления в визуальном дереве. Другими словами, элемент, который был объявлен раньше, будет показан раньше.

11.1. Панель StackPanel

Панель System.Windows.Controls.StackPanel организует помещенные в нее элементы в горизонтальный или вертикальный стек, т.е. выстраивает их друг за другом.

Свойства класса System.Windows.Controls.StackPanel:

- **Orientation** (System.Windows.Controls.Orientation). Получает или задает режим упорядочивания дочерних

элементов. Значение по умолчанию — System.Windows.Controls.Orientation.Vertical.

Для того, чтобы указать режим упорядочивания элементов необходимо использовать перечисление System.Windows.Controls.Orientation, которое содержит следующие варианты:

- Horizontal. Панель упорядочивает элементы в горизонтальный стек.
- Vertical. Панель упорядочивает элементы в вертикальный стек.

Вертикально ориентированная панель, располагает помещенные в нее элементы сверху вниз, в порядке их объявления в разметке. Высота каждого элемента задается значением необходимым для отображения его содержимого. При этом в ширину элементы растягиваются на всю панель.

Приведенный ниже фрагмент разметки демонстрирует вертикально ориентированную панель (полный пример находится в папке Wpf.Panels.StackPanel.Vertical.Xaml):

XAML

```
<StackPanel>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
    <Button>Button 5</Button>
</StackPanel>
```

Результат приведенной выше разметки показан на рис. 21.

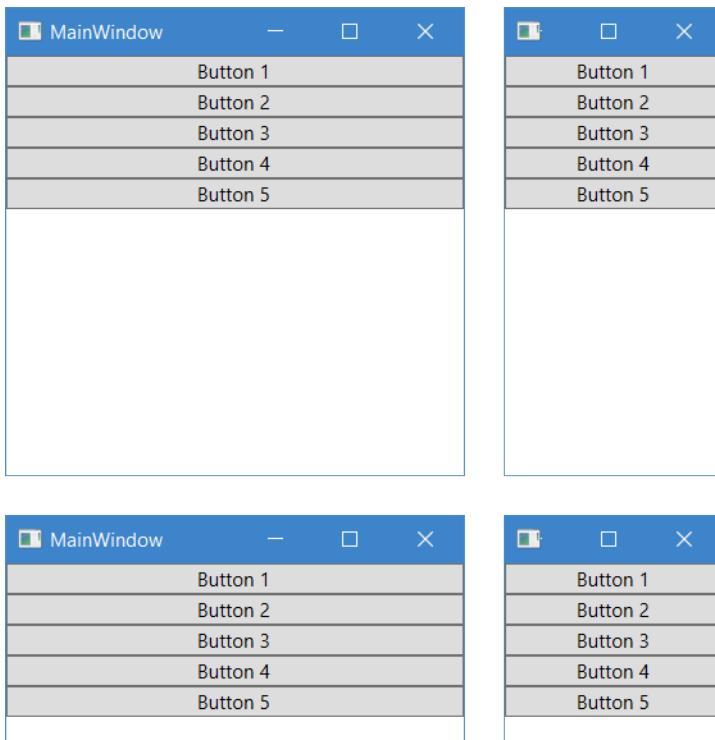


Рис. 21. Компоновка элементов управления с применением вертикально ориентированной панели StackPanel

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Panels.StackPanel.Vertical.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var stackPanel = new StackPanel();
stackPanel.Children.Add(button1);
```

```
stackPanel.Children.Add(button2);  
stackPanel.Children.Add(button3);  
stackPanel.Children.Add(button4);  
stackPanel.Children.Add(button5);
```

По умолчанию панель `System.Windows.Controls.StackPanel` упорядочивает элементы в вертикальный стек. Для того, чтобы изменить вид упорядочивания с вертикального на горизонтальный, необходимо задать свойству `System.Windows.Controls.StackPanel.Orientation` значение `System.Windows.Controls.Orientation.Horizontal`.

Горизонтально ориентированная панель, располагает помещенные в нее элементы слева направо, в порядке их объявления в разметке. Ширина каждого элемента задается значением необходимым для отображения его содержимого. При этом в высоту элементы растягиваются на всю панель.

Приведенный ниже фрагмент разметки демонстрирует горизонтально ориентированную панель (полный пример находится в папке `Wpf.Panels.StackPanel.Horizontal.Xaml`):

XAML

```
<StackPanel Orientation="Horizontal">  
    <Button>Button 1</Button>  
    <Button>Button 2</Button>  
    <Button>Button 3</Button>  
    <Button>Button 4</Button>  
    <Button>Button 5</Button>  
</StackPanel>
```

Результат приведенной выше разметки показан на рис. 22.

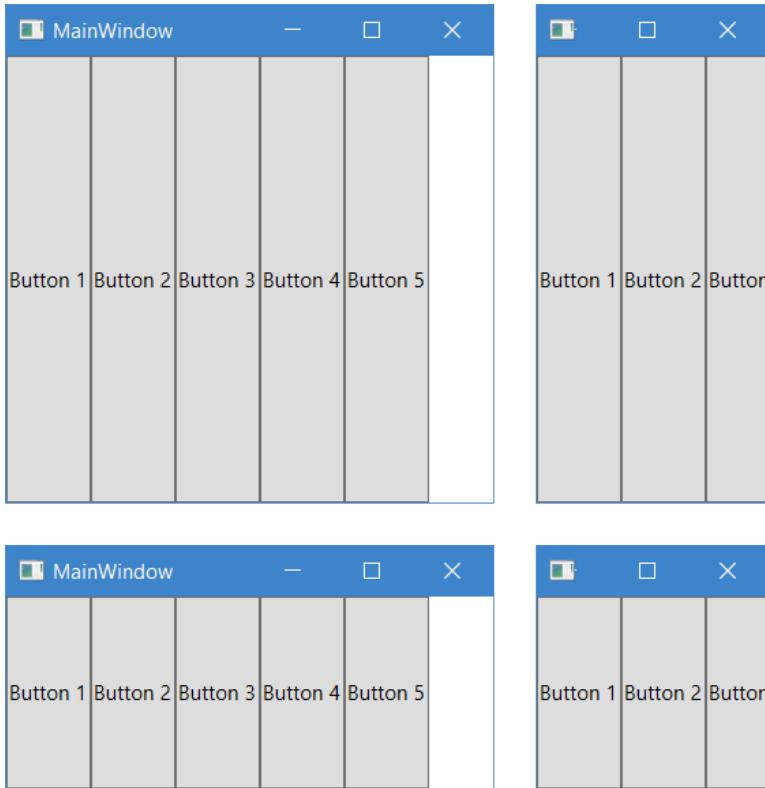


Рис. 22. Компоновка элементов управления с применением горизонтально ориентированной панели StackPanel

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Panels.StackPanel.Horizontal.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
```

```
var stackPanel = new StackPanel { Orientation =
    Orientation.Horizontal };
stackPanel.Children.Add(button1);
stackPanel.Children.Add(button2);
stackPanel.Children.Add(button3);
stackPanel.Children.Add(button4);
stackPanel.Children.Add(button5);
```

11.2. Панель WrapPanel

Панель System.Windows.Controls.WrapPanel организует помещенные в нее элементы в горизонтальный или вертикальный стек, но, в отличии от панели System.Windows.Controls.StackPanel, если элементы не помещаются в пределах доступного пространства панели, то, не поместившиеся переносятся на другую строку или колонку.

Свойства класса System.Windows.Controls.WrapPanel:

- **Orientation** (System.Windows.Controls.Orientation). Получает или задает режим упорядочивания дочерних элементов. Значение по умолчанию — System.Windows.Controls.Orientation.Horizontal.

Горизонтально ориентированная панель, располагает помещенные в нее элементы слева направо, в порядке их объявления в разметке. Ширина и высота каждого элемента задается значением необходимым для отображения его содержимого. При этом если в одной строке присутствуют элементы с разной высотой, то все будут растянуты до максимальной из существующих.

Приведенный ниже фрагмент разметки демонстрирует горизонтально ориентированную панель (полный пример находится в папке Wpf.Panels.WrapPanel.Horizontal.Xaml):

XAML

```
<WrapPanel>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
    <Button>Button 5</Button>
</WrapPanel>
```

Результат приведенной выше разметки показан на рис. 23.

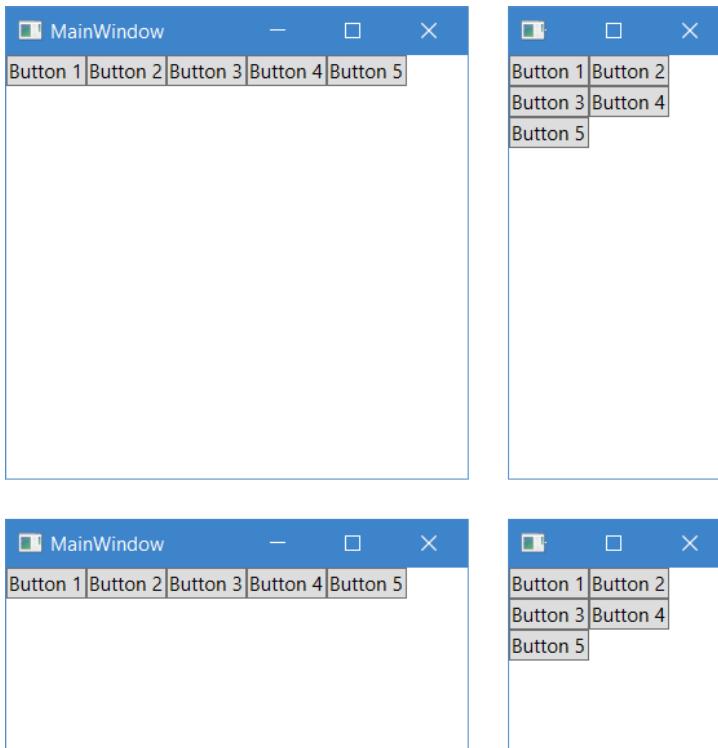


Рис. 23. Компоновка элементов управления с применением горизонтально ориентированной панели WrapPanel

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Panels.WrapPanel.Horizontal.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };

var wrapPanel = new WrapPanel();
wrapPanel.Children.Add(button1);
wrapPanel.Children.Add(button2);
wrapPanel.Children.Add(button3);
wrapPanel.Children.Add(button4);
wrapPanel.Children.Add(button5);
```

По умолчанию панель System.Windows.Controls.WrapPanel упорядочивает элементы в горизонтальный стек. Для того, чтобы изменить вид упорядочивания с горизонтального на вертикальный, необходимо задать свойству System.Windows.Controls.WrapPanel.Orientation значение System.Windows.Controls.Orientation.Vertical.

Вертикально ориентированная панель, располагает помещенные в нее элементы сверху вниз, в порядке их объявления в разметке. Ширина и высота каждого элемента задается значением необходимым для отображения его содержимого. При этом если в одной колонке присутствуют элементы с разной шириной, то все будут растянуты до максимальной из существующих.

Приведенный ниже фрагмент разметки демонстрирует вертикально ориентированную панель (полный пример находится в папке Wpf.Panels.WrapPanel.Vertical.Xaml):

XAML

```
<WrapPanel Orientation="Vertical">
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
    <Button>Button 5</Button>
</WrapPanel>
```

Результат приведенной выше разметки показан на рис. 24.

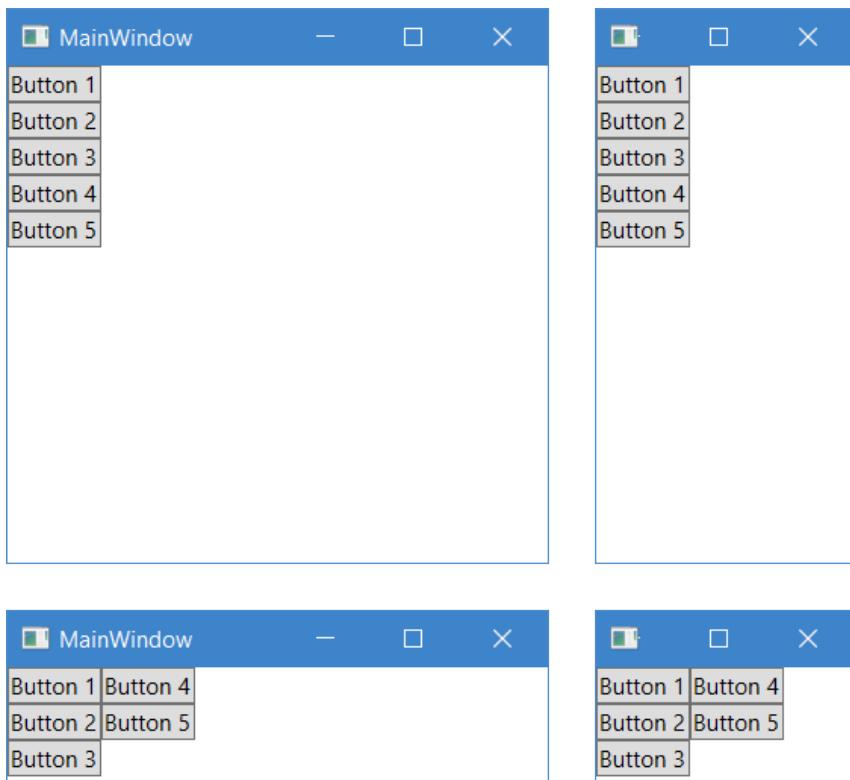


Рис. 24. Компоновка элементов управления с применением вертикально ориентированной панели WrapPanel

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Panels.WrapPanel.Vertical.CSharp):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };

var wrapPanel = new WrapPanel { Orientation =
    Orientation.Vertical };

wrapPanel.Children.Add(button1);
wrapPanel.Children.Add(button2);
wrapPanel.Children.Add(button3);
wrapPanel.Children.Add(button4);
wrapPanel.Children.Add(button5);
```

11.3. Панель DockPanel

Панель System.Windows.Controls.DockPanel организовывает помещенные в нее элементы, пристыковывая их одной из своих границ.

Свойства класса System.Windows.Controls.DockPanel:

- **LastChildFill** (System.Boolean). Получает или задает значение, которое указывает, должен ли последний помещенный в панель элемент растягиваться на все оставшееся незанятое пространство. **true** если последний элемент должен занимать все оставшееся пространство; иначе — **false**. Значение по умолчанию — **true**.

Присоединенные свойства класса System.Windows.Controls.DockPanel:

- **Dock** (System.Windows.Controls.Dock). Получает или задает сторону панели, к которой следует пристыковать элемент.

Элементы, помещенные в панель, пристыковываются в порядке своего объявления. Каждый следующий элемент используется только то пространство, которое осталось после пристыковки предыдущего. Для того, чтобы задать вид пристыковки элементу, необходимо использовать присоединенной свойство панели — System.Windows.Controls.DockPanel.Dock.

Приведенный ниже фрагмент разметки демонстрирует панель (полный пример находится в папке Wpf.Panels.DockPanel.Xaml):

XAML

```
<DockPanel>
    <Button DockPanel.Dock="Top">Button 1</Button>
    <Button DockPanel.Dock="Bottom">Button 2</Button>
    <Button DockPanel.Dock="Left">Button 3</Button>
    <Button DockPanel.Dock="Top">Button 4</Button>
    <Button>Button 5</Button>
</DockPanel>
```

Результат приведенной выше разметки показан на рис. 25.

Получившийся результат может заставить задуматься, почему обе кнопки, пристыкованные к верхней стороне панели выглядят по-разному. Для того, чтобы ответить

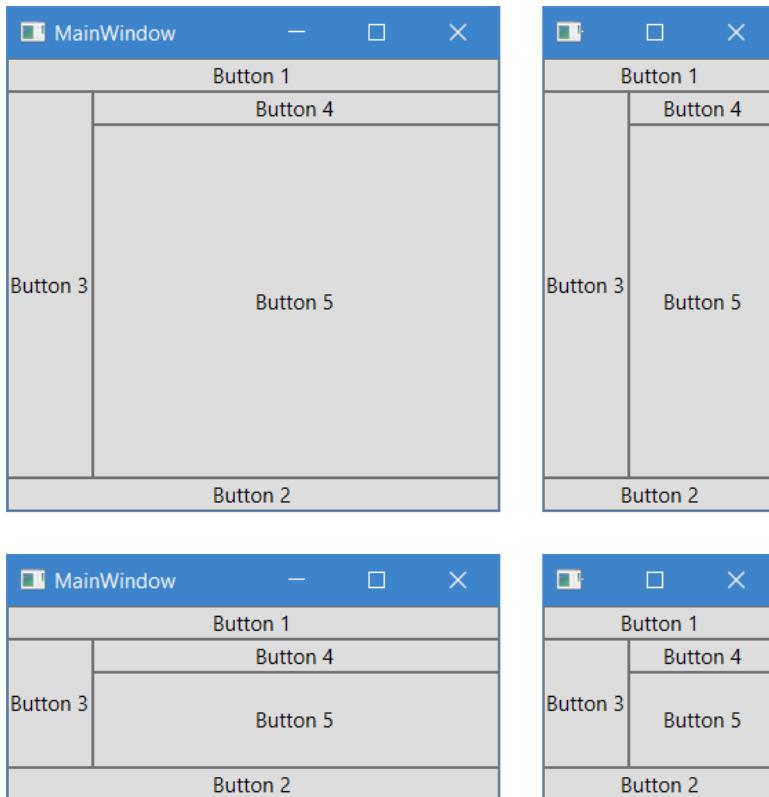


Рис. 25. Компоновка элементов управления с применением панели DockPanel

на этот вопрос, давайте рассмотрим алгоритм работы данной панели.

Как было сказано ранее, при вычислении размера и расположения элементов, панель обрабатывает их в порядке объявления, т.е. вначале будет обработана первая кнопка. Она должна быть пристыкована к верхней границе панели и при этом она может использовать все пространство панели. Пристыковываясь к верхней стороне панели элемент помещается как можно выше (к самой границе панели

или под другие пристыкованные ранее к верху элементы), занимая всю доступную ширину. Размер высоты вычисляется минимально возможным, т.е. под содержимое. Вторая кнопка пристыковывается к нижней панели точно по такому же принципу (растягивается на всю ширину панели и на высоту содержимого). Третья кнопка должна быть пристыкована к левой стороне. Когда элемент стыкуется с левой или правой границей панели, он растягивается в высоту на все свободное пространство, а в ширину по содержимому. Поэтому пристыковываясь к левой стороне, кнопка растянулась в высоту не на весь размер панели, а только на высоту оставшейся (не использованной) области. Четвертая кнопка, пристыковываясь к верхней стороне панели, помещается под первую и также растягивается на всю доступную ширину. В этом случае у нее также не получается занять ширину всей панели, так как часть области в этом месте уже отдана под использование третьей кнопкой. Последняя, пятая, кнопка занимает все оставшееся не использованным пространство панели, за счет того, что свойство панели `System.Windows.Controls.DockPanel.LastChildFill` по умолчанию имеет значение `true`.

Рассмотренной выше разметке соответствует следующий код (полный пример находится в `Wpf.Panels.DockPanel.CSharp`):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
```

```
var dockPanel = new DockPanel();
dockPanel.Children.Add(button1);
dockPanel.Children.Add(button2);
dockPanel.Children.Add(button3);
dockPanel.Children.Add(button4);
dockPanel.Children.Add(button5);
DockPanel.SetDock(button1, Dock.Top);
DockPanel.SetDock(button2, Dock.Bottom);
DockPanel.SetDock(button3, Dock.Left);
DockPanel.SetDock(button4, Dock.Top);
```

11.4. Панель Grid

Панель `System.Windows.Controls.Grid` организовывает помещенные в нее элементы, в виде сетки с настраиваемыми размерами строк и колонок.

Свойства класса `System.Windows.Controls.Grid`:

- **ColumnDefinitions** (`System.Windows.Controls.ColumnDefinitionCollection`). Получает коллекцию колонок сетки. Значение по умолчанию — пустая коллекция.
- **RowDefinitions** (`System.Windows.Controls.RowDefinitionCollection`). Получает коллекцию строк сетки. Значение по умолчанию — пустая коллекция.
- **ShowGridLines** (`System.Boolean`). Получает или задает значение, которое указывает, должна ли отображаться сетка. `true` если сетка должна отображаться; иначе — `false`. Значение по умолчанию — `false`.

Присоединенные свойства класса `System.Windows.Controls.Grid`:

- **Column** (`System.Int32`). Получает или задает индекс колонки сетки, в которой должен находиться элемент.

Для того, чтобы поместить элемент в первую колонку необходимо указать значение 0, во вторую — 1, и т.д.

- **ColumnSpan** (System.Int32). Получает или задает количество колонок, которое должен занимать элемент, начиная с той, в которой он располагается.
- **Row** (System.Int32). Получает или задает индекс строки сетки, в которой должен находиться элемент. Для того, чтобы поместить элемент в первую строку необходимо указать значение 0, во вторую — 1, и т.д.
- **RowSpan** (System.Int32). Получает или задает количество строк, которое должен занимать элемент, начиная с той, в которой он располагается.

При описании панели необходимо задать количество строк и колонок результирующей сетки. При необходимости можно также задать высоту для каждой строки и ширину для каждой колонки. Элементы, помещенные в панель, располагаются в ячейках, образованных пересечением строк и колонок. Если пропустить описание строк, то одна строка все равно будет присутствовать в сетке. Аналогичная ситуация происходит и с колонками. Таким образом, если пропустить объявление строк и колонок полностью, то сетка будет состоять только из одной ячейки.

Ячейки нельзя объединять между собой, но элементам, помещенным в панель можно задать параметр, отвечающий за то, сколько строк и/или колонок он должен занимать.

Если несколько элементов поместить в одну и ту же ячейку, то они будут перекрываться. В этом случае элементы будут отображаться в порядке своего объявления,

т.е. объявленный позже будет находиться поверх того, который был объявлен раньше.

Для того, чтобы задать элементу ячейку, в которую он должен быть помещен, необходимо использовать присоединенные свойства панели: System.Windows.Controls.Grid.Row и System.Windows.Controls.Grid.Column. Если какое-то из них (или оба) не задать, то будет взято значение 0 как для строки, так и для колонки.

Для описания строк используется класс System.Windows.Controls.RowDefinition, а для описания колонок — System.Windows.Controls.ColumnDefinition.

Свойства класса System.Windows.Controls.RowDefinition:

- **ActualHeight** (System.Double). Получает рассчитанную высоту строки после двух этапов компоновки. Значение по умолчанию — 0.0.
- **Height** (System.Windows.Controls.GridLength). Получает или задает высоту строки. Значение по умолчанию — 1.0.
- **MaxHeight** (System.Double). Получает или задает максимально возможную высоту строки. Значение по умолчанию — System.Double.PositiveInfinity.
- **MinHeight** (System.Double). Получает или задает минимально возможную высоту строки. Значение по умолчанию — 0.0.

Свойства класса System.Windows.Controls.ColumnDefiniton:

- **ActualWidth** (System.Double). Получает рассчитанную ширину колонки после двух этапов компоновки. Значение по умолчанию — 0.0.

- **MaxWidth** (System.Double). Получает или задает максимально возможную ширину колонки. Значение по умолчанию — System.Double.PositiveInfinity.
- **MinWidth** (System.Double). Получает или задает минимально возможную ширину колонки. Значение по умолчанию — 0.0.
- **Width** (System.Windows.Controls.GridLength). Получает или задает ширину колонки. Значение по умолчанию — 1.0.

Приведенный ниже фрагмент разметки демонстрирует вариант сетки, в которой пространство между колонками и строками делится поровну (полный пример находится в папке Wpf.Panels.Grid.Uniform.Xaml):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Column="0" Grid.Row="0">1</Button>
    <Button Grid.Column="1" Grid.Row="0">2</Button>
    <Button Grid.Column="2" Grid.Row="0">3</Button>
    <Button Grid.Column="3" Grid.Row="0">4</Button>
    <Button Grid.Column="0" Grid.Row="1">5</Button>
    <Button Grid.Column="1" Grid.Row="1">6</Button>
```

```
<Button Grid.Column="2" Grid.Row="1">7</Button>
<Button Grid.Column="3" Grid.Row="1">8</Button>
<Button Grid.Column="0" Grid.Row="2">9</Button>
<Button Grid.Column="1" Grid.Row="2">10</Button>
<Button Grid.Column="2" Grid.Row="2">11</Button>
<Button Grid.Column="3" Grid.Row="2">12</Button>
</Grid>
```

Результат приведенной выше разметки показан на рис. 26.

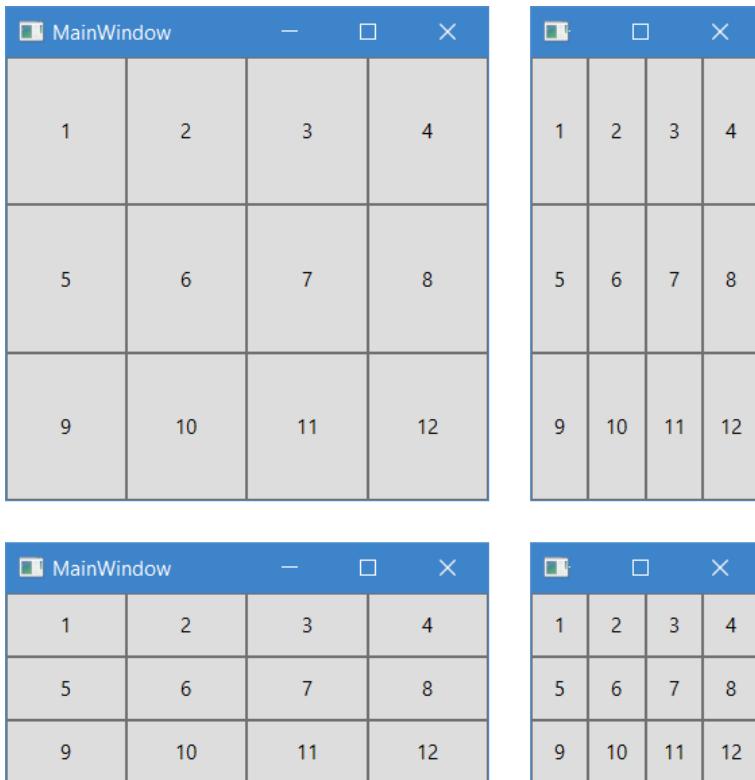


Рис. 26. Компоновка элементов управления с применением панели Grid.

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Panels.Grid.Uniform.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };
var button6 = new Button { Content = 6 };
var button7 = new Button { Content = 7 };
var button8 = new Button { Content = 8 };
var button9 = new Button { Content = 9 };
var button10 = new Button { Content = 10 };
var button11 = new Button { Content = 11 };
var button12 = new Button { Content = 12 };

var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);
grid.Children.Add(button5);
grid.Children.Add(button6);
grid.Children.Add(button7);
grid.Children.Add(button8);
grid.Children.Add(button9);
```

```
grid.Children.Add(button10);
grid.Children.Add(button11);
grid.Children.Add(button12);
Grid.SetColumn(button1, 0);
Grid.SetRow(button1, 0);
Grid.SetColumn(button2, 1);
Grid.SetRow(button2, 0);
Grid.SetColumn(button3, 2);
Grid.SetRow(button3, 0);
Grid.SetColumn(button4, 3);
Grid.SetRow(button4, 0);
Grid.SetColumn(button5, 0);
Grid.SetRow(button5, 1);
Grid.SetColumn(button6, 1);
Grid.SetRow(button6, 1);
Grid.SetColumn(button7, 2);
Grid.SetRow(button7, 1);
Grid.SetColumn(button8, 3);
Grid.SetRow(button8, 1);
Grid.SetColumn(button9, 0);
Grid.SetRow(button9, 2);
Grid.SetColumn(button10, 1);
Grid.SetRow(button10, 2);
Grid.SetColumn(button11, 2);
Grid.SetRow(button11, 2);
Grid.SetColumn(button12, 3);
Grid.SetRow(button12, 2);
```

При описании колонок и строк можно не указывать явным образом какой шириной и высотой они должны обладать. В этом случае вся доступная ширина панели будет поделена поровну на колонки, а высота на строки. Однако, бывает необходимо задать эти параметры согласно какому-то критерию.

Размерность каждой колонки и строки может быть задана одним из следующих способов:

- **Явное указание.** В этом случае ширина колонки или высота строки будет равна заданной величине.
- **Автоматическое вычисление.** В этом случае ширина колонки или высота строки будет минимально возможной, чтобы полностью вместить свое содержимое (*size to content*).
- **Указание соотношений.** В этом случае нескольким колонкам или строкам указывается каким образом их ширина или высота должна соотноситься между собой. Например, 1:2 или 3:5.

Для описания размерности колонки или строки используется структура System.Windows.GridLength.

Конструкторы структуры System.Windows.GridLength:

- **GridLength(pixels).** Инициализирует экземпляр структуры указанным количеством аппаратно-независимых единиц.
- **GridLength(value, type).** Инициализирует экземпляр структуры указанным значением и видом этого значения.

Статические свойства структуры System.Windows.GridLength:

- **Auto** (System.Windows.GridLength). Получает объект, описывающий автоматически вычисляемую размерность.

Свойства структуры System.Windows.GridLength:

- **GridUnitType** (System.Windows.GridUnitType). Получает значение, описывающее вид значения размерности, которая содержится в объекте.

- **IsAbsolute** (System.Boolean). Получает значение, которое указывает, является ли значение, хранимое в объекте величиной в аппаратно-независимых единицах. **true** если значение свойства System.Windows.GridLength.GridUnitType равно System.Windows.GridUnitType.Pixel; иначе — **false**.
- **IsAuto** (System.Boolean). Получает значение, которое указывает, является ли значение, хранимое в объекте автоматически вычислимой величиной, в зависимости от содержимого. **true** если значение свойства System.Windows.GridLength.GridUnitType равно System.Windows.GridUnitType.Auto; иначе — **false**.
- **IsStar** (System.Boolean). Получает значение, которое указывает, является ли значение, хранимое в объекте указанием соотношения. **true** если значение свойства System.Windows.GridLength.GridUnitType равно System.Windows.GridUnitType.Star; иначе — **false**.
- **Value** (System.Double). Получает значение, хранимое в объекте.

Приведенный ниже фрагмент разметки демонстрирует вариант сетки с разными вариантами указания размерности колонкам и строкам (полный пример находится в папке Wpf.Panels.Grid.NonUniform.Xaml):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="30"/>
        <ColumnDefinition Width="Auto"/>
```

```

    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="2*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="40"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="2*"/>
    <RowDefinition Height="3*"/>
</Grid.RowDefinitions>

<Button Grid.Column="0" Grid.Row="0">1</Button>
<Button Grid.Column="1" Grid.Row="0">2</Button>
<Button Grid.Column="2" Grid.Row="0">3</Button>
<Button Grid.Column="3" Grid.Row="0">4</Button>
<Button Grid.Column="0" Grid.Row="1">5</Button>
<Button Grid.Column="1" Grid.Row="1">6</Button>
<Button Grid.Column="2" Grid.Row="1">7</Button>
<Button Grid.Column="3" Grid.Row="1">8</Button>
<Button Grid.Column="0" Grid.Row="2">9</Button>
<Button Grid.Column="1" Grid.Row="2">10</Button>
<Button Grid.Column="2" Grid.Row="2">11</Button>
<Button Grid.Column="3" Grid.Row="2">12</Button>
<Button Grid.Column="0" Grid.Row="3">13</Button>
<Button Grid.Column="1" Grid.Row="3">14</Button>
<Button Grid.Column="2" Grid.Row="3">15</Button>
<Button Grid.Column="3" Grid.Row="3">16</Button>
</Grid>

```

Результат приведенной выше разметки показан на рис. 27.

В разметке выше продемонстрированы все три формы указания размерности:

- **Явное указание.** В этом случае в качестве значения указывается число, означающее количество аппаратно-независимых единиц.

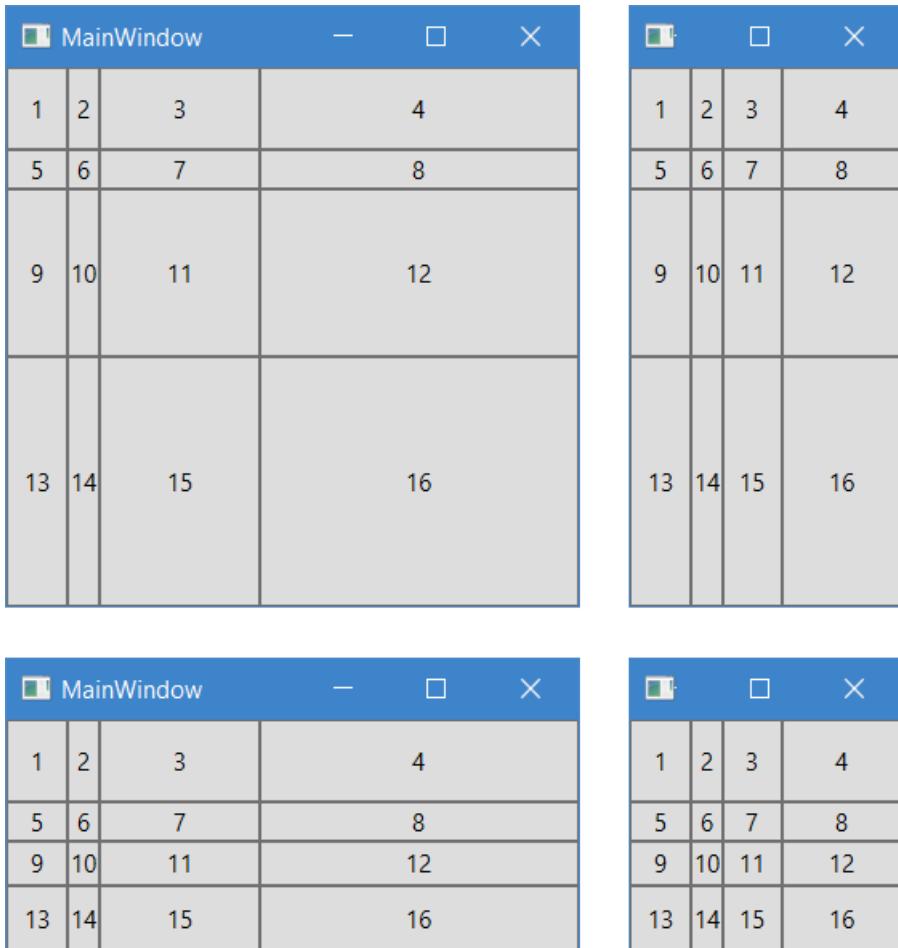


Рис. 27. Компоновка элементов управления с применением панели Grid

- **Автоматическое вычисление.** В этом случае в качестве значения указывается значение **Auto**.
- **Указание соотношений.** В этом случае в качестве значения указывается число и звездочка (в случае если число равняется 1, то его можно опустить, т.е. * то же,

что и 1^*). Например, если у одной колонки указано значение 2^* , а у другой 3^* , то они соотносятся как 2:3.

Если не задать значение высоты или ширины какой-то колонке или строке, то значение будет установлено равным $*$.

В первую очередь пространство панели разделяется между колонками и строками, для которых размер был указан фиксированный или «под содержимое». После этого все оставшееся пространство делится согласно указанным соотношениям.

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Panels.Grid.NonUniform.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };
var button6 = new Button { Content = 6 };
var button7 = new Button { Content = 7 };
var button8 = new Button { Content = 8 };
var button9 = new Button { Content = 9 };
var button10 = new Button { Content = 10 };
var button11 = new Button { Content = 11 };
var button12 = new Button { Content = 12 };
var button13 = new Button { Content = 13 };
var button14 = new Button { Content = 14 };
var button15 = new Button { Content = 15 };
var button16 = new Button { Content = 16 };

var grid = new Grid();
```

```
grid.ColumnDefinitions.Add(new ColumnDefinition
    { Width = new GridLength(30.0) });
grid.ColumnDefinitions.Add(new ColumnDefinition
    { Width = GridLength.Auto });
grid.ColumnDefinitions.Add(
    new ColumnDefinition { Width =
    new GridLength(1.0, GridUnitType.Star) });
grid.ColumnDefinitions.Add(
    new ColumnDefinition { Width =
    new GridLength(2.0, GridUnitType.Star) });
grid.RowDefinitions.Add(new RowDefinition { Height =
    new GridLength(40.0) });
grid.RowDefinitions.Add(new RowDefinition { Height =
    GridLength.Auto });
grid.RowDefinitions.Add(
    new RowDefinition { Height =
    new GridLength(2.0, GridUnitType.Star) });
grid.RowDefinitions.Add(
    new RowDefinition { Height =
    new GridLength(3.0, GridUnitType.Star) });

grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);
grid.Children.Add(button5);
grid.Children.Add(button6);
grid.Children.Add(button7);
grid.Children.Add(button8);
grid.Children.Add(button9);
grid.Children.Add(button10);
grid.Children.Add(button11);
grid.Children.Add(button12);
grid.Children.Add(button13);
```

```
grid.Children.Add(button14);
grid.Children.Add(button15);
grid.Children.Add(button16);

Grid.SetColumn(button1, 0);
Grid.SetRow(button1, 0);
Grid.SetColumn(button2, 1);
Grid.SetRow(button2, 0);
Grid.SetColumn(button3, 2);
Grid.SetRow(button3, 0);
Grid.SetColumn(button4, 3);
Grid.SetRow(button4, 0);
Grid.SetColumn(button5, 0);
Grid.SetRow(button5, 1);
Grid.SetColumn(button6, 1);
Grid.SetRow(button6, 1);
Grid.SetColumn(button7, 2);
Grid.SetRow(button7, 1);
Grid.SetColumn(button8, 3);
Grid.SetRow(button8, 1);
Grid.SetColumn(button9, 0);
Grid.SetRow(button9, 2);
Grid.SetColumn(button10, 1);
Grid.SetRow(button10, 2);
Grid.SetColumn(button11, 2);
Grid.SetRow(button11, 2);
Grid.SetColumn(button12, 3);
Grid.SetRow(button12, 2);
Grid.SetColumn(button13, 0);
Grid.SetRow(button13, 3);
Grid.SetColumn(button14, 1);
Grid.SetRow(button14, 3);
Grid.SetColumn(button15, 2);
Grid.SetRow(button15, 3);
Grid.SetColumn(button16, 3);
Grid.SetRow(button16, 3);
```

В некоторых ситуациях необходимо, чтобы элемент, помещенный в панель занимал больше одной строки и/или колонки. В таких ситуациях необходимо использовать присоединенные свойства панели: `System.Windows.Controls.Grid.RowSpan` и `System.Windows.Controls.Grid.ColumnSpan`.

Каждому элементу эти свойства задаются отдельно. Если более одного элемента находятся в одной и той же ячейки, и одному из них задать эти свойства для того, чтобы он растянулся, то на другие элементы в этой ячейке это никак не повлияет.

Приведенный ниже фрагмент разметки демонстрирует вариант сетки с элементами, занимающими более одной ячейки (полный пример находится в папке `Wpf.Panels.Grid.Span.Xaml`):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Grid.Column="0" Grid.ColumnSpan="2"
            Grid.Row="0">1</Button>
    <Button Grid.Column="2" Grid.Row="0"
            Grid.RowSpan="2">2</Button>
    <Button Grid.Column="0" Grid.Row="1">3</Button>
```

```
<Button Grid.Column="1" Grid.Row="1">4</Button>
<Button Grid.Column="0" Grid.ColumnSpan="3"
        Grid.Row="2">5</Button>
</Grid>
```

Результат приведенной выше разметки показан на рис. 28.

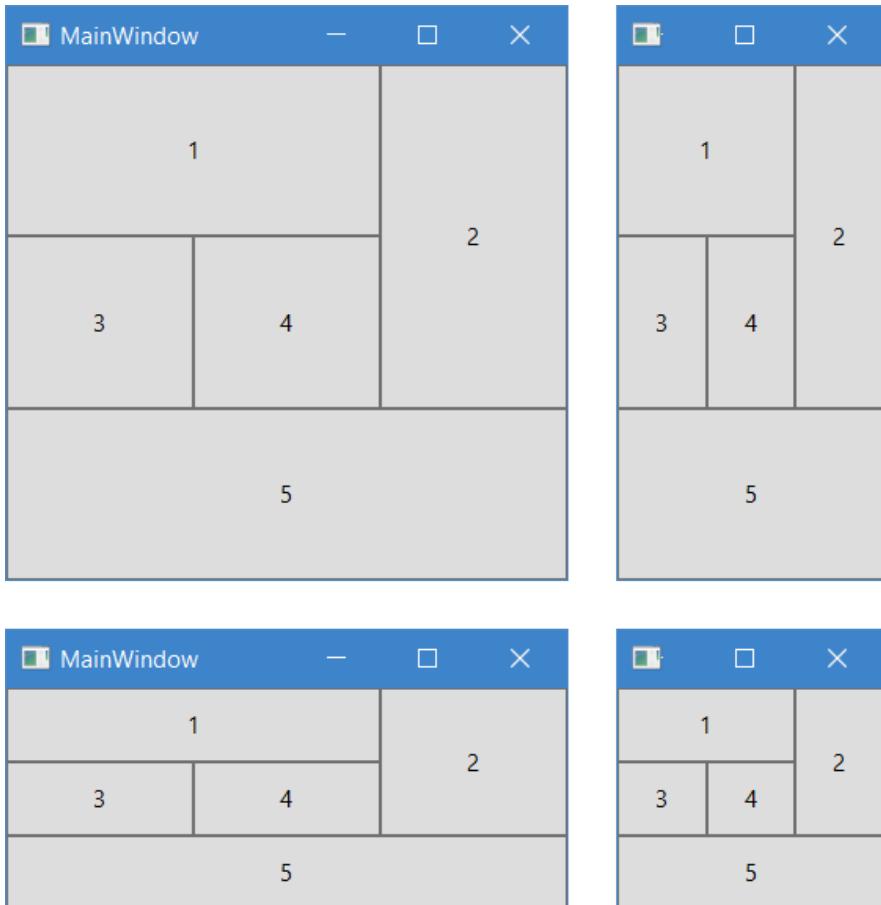


Рис. 28. Компоновка элементов управления с применением панели Grid

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Panels.Grid.Span.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };

var grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());

grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());

grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);
grid.Children.Add(button5);

Grid.SetColumn(button1, 0);
Grid.SetRow(button1, 0);
Grid.SetColumn(button2, 2);
Grid.SetRow(button2, 0);
Grid.SetColumn(button3, 0);
Grid.SetRow(button3, 1);
Grid.SetColumn(button4, 1);
Grid.SetRow(button4, 1);
Grid.SetColumn(button5, 0);
```

```
Grid.SetRow(button5, 2);
Grid.SetColumnSpan(button1, 2);
Grid.SetColumnSpan(button5, 3);
Grid.SetRowSpan(button2, 2);
```

11.5. Панель UniformGrid

Панель `System.Windows.Controls.Primitives.UniformGrid` организовывает помещенные в нее элементы, в виде сетки с одинаковыми размерами строк и колонок. Эта панель является упрощенной версией панели `System.Windows.Controls.Grid`.

Свойства класса `System.Windows.Controls.Primitives.UniformGrid`:

- **Columns** (`System.Int32`). Получает или задает количество колонок в сетке. Значение по умолчанию — 0.
- **FirstColumn** (`System.Int32`). Получает или задает количество пустых ячеек, находящихся в начале первой строки. Значение по умолчанию — 0.
- **Rows** (`System.Int32`). Получает или задает количество строк в сетке. Значение по умолчанию — 0.

Приведенный ниже фрагмент разметки демонстрирует сетку (полный пример находится в папке `Wpf.Panels.UniformGrid.Xaml`):

XAML

```
<UniformGrid Columns="3" Rows="2">
    <Button>1</Button>
    <Button>2</Button>
    <Button>3</Button>
```

```
<Button>4</Button>
<Button>5</Button>
<Button>6</Button>
</UniformGrid>
```

Результат приведенной выше разметки показан на рис. 29.

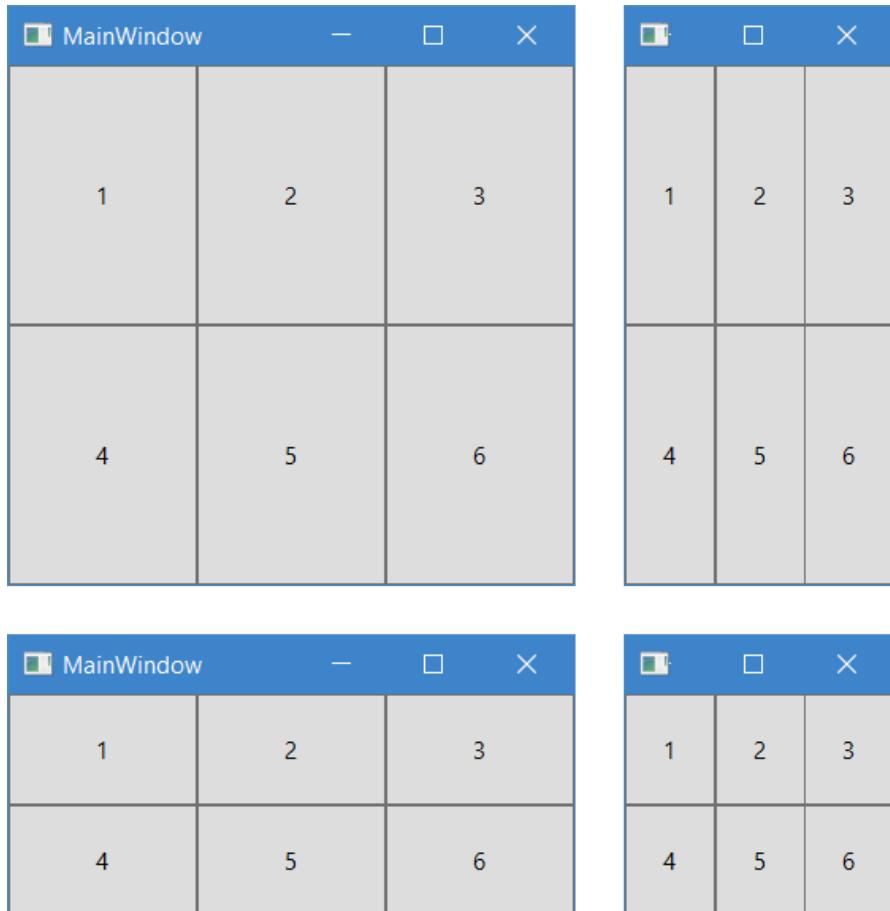


Рис. 29. Компоновка элементов управления с применением панели UniformGrid

Рассмотренной разметке соответствует следующий код (полный пример находится в Wpf.Panels.UniformGrid.CSharp):

C#

```
var button1 = new Button { Content = 1 };
var button2 = new Button { Content = 2 };
var button3 = new Button { Content = 3 };
var button4 = new Button { Content = 4 };
var button5 = new Button { Content = 5 };
var button6 = new Button { Content = 6 };
var uniformGrid = new UniformGrid { Columns = 3,
    Rows = 2 };

uniformGrid.Children.Add(button1);
uniformGrid.Children.Add(button2);
uniformGrid.Children.Add(button3);
uniformGrid.Children.Add(button4);
uniformGrid.Children.Add(button5);
uniformGrid.Children.Add(button6);
```

11.6. Панель Canvas

Панель System.Windows.Controls.Canvas организовывает помещенные в нее элементы согласно точно заданным координатам.

Присоединенные свойства класса System.Windows.Controls.Canvas:

- **Bottom** (System.Double). Получает или задает отступ от нижней границы панели до элемента. Значение задается в аппаратно-независимых единицах.
- **Left** (System.Double). Получает или задает отступ от левой границы панели до элемента. Значение задается в аппаратно-независимых единицах.

- **Right** (System.Double). Получает или задает отступ от правой границы панели до элемента. Значение задается в аппаратно-независимых единицах.
- **Top** (System.Double). Получает или задает отступ от верхней границы панели до элемента. Значение задается в аппаратно-независимых единицах.

Если не задавать элементу горизонтальный отступ, то он будет выравнен по левому краю панели, т.е. то же самое, что задать отступ слева равный 0.0. Если не задавать элементу вертикальный отступ, то он будет выравнен по верхнему краю панели, т.е. то же самое, что задать отступ сверху равный 0.0.

Если задать элементу взаимоисключающие отступы, то отступы System.Windows.Controls.Canvas.Left и System.Windows.Controls.Canvas.Top будут приоритетнее.

Приведенный ниже фрагмент разметки демонстрирует панель (полный пример находится в папке Wpf.Panels.Canvas.Xaml):

XAML

```
<Canvas>
    <Button>Button 1</Button>
    <Button Canvas.Left="70">Button 2</Button>
    <Button Canvas.Top="70">Button 3</Button>
    <Button Canvas.Bottom="70">Button 4</Button>
    <Button Canvas.Right="70">Button 5</Button>
    <Button Canvas.Left="70" Canvas.Top="70">Button 6
        </Button>
</Canvas>
```

Результат приведенной выше разметки показан на рис. 30.

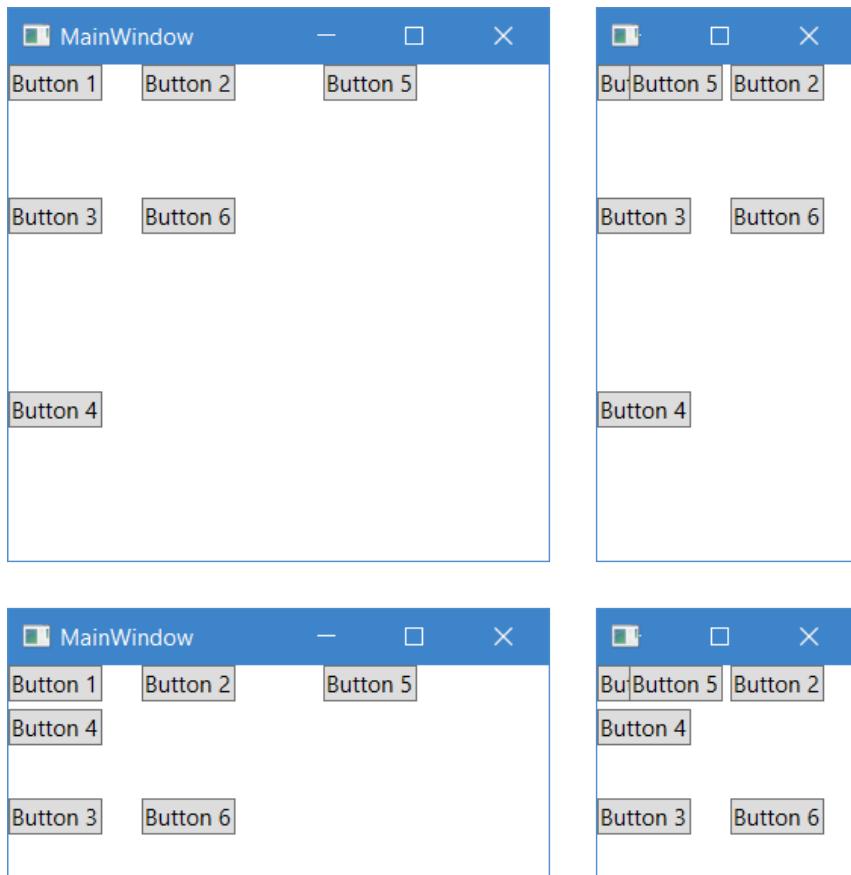


Рис. 30. Компоновка элементов управления с применением панели Canvas

Рассмотренной выше разметке соответствует следующий код (полный пример находится в `Wpf.Panels.Canvas.CSharp`):

C#

```
var button1 = new Button { Content = "Button 1" };
var button2 = new Button { Content = "Button 2" };
var button3 = new Button { Content = "Button 3" };
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var button6 = new Button { Content = "Button 6" };
```

```
var button4 = new Button { Content = "Button 4" };
var button5 = new Button { Content = "Button 5" };
var button6 = new Button { Content = "Button 6" };

var canvas = new Canvas();
canvas.Children.Add(button1);
canvas.Children.Add(button2);
canvas.Children.Add(button3);
canvas.Children.Add(button4);
canvas.Children.Add(button5);
canvas.Children.Add(button6);

Canvas.SetLeft(button2, 70.0);
Canvas.SetTop(button3, 70.0);
Canvas.SetBottom(button4, 70.0);
Canvas.SetRight(button5, 70.0);
Canvas.SetLeft(button6, 70.0);
Canvas.SetTop(button6, 70.0);
```

12. Позиционирование элементов

Практически все элементы, находящиеся в визуальном дереве, наследуются от класса `System.Windows.FrameworkElement`. В данном разделе будут рассмотрены свойства этого класса, которые используются для более точного позиционирования и управления размерами визуальных элементов.

Свойства класса `System.Windows.FrameworkElement`, связанные с позиционированием:

- **ActualHeight** (`System.Double`). Получает действительную (рассчитанную) высоту элемента в аппаратно-независимых единицах. Значение по умолчанию — 0.0.
- **ActualWidth** (`System.Double`). Получает действительную (рассчитанную) ширину элемента в аппаратно-независимых единицах. Значение по умолчанию — 0.0.
- **Height** (`System.Double`). Получает или задает предложенную высоту элемента в аппаратно-независимых единицах. Данное значение должно быть больше или равно 0.0. Значение по умолчанию — `System.Double.NaN`.
- **HorizontalAlignment** (`System.Windows.HorizontalAlignment`). Получает или задает горизонтальное выравнивание элемента для тех случаев, когда он помещается внутрь другого элемента (например, панели). Значение по умолчанию — `System.Windows.HorizontalAlignment.Stretch`.

- **Margin** (`System.Windows.Thickness`). Получает или задает внешние отступы для элемента. Значение по умолчанию — объект `System.Windows.Thickness`, описывающий нулевые отступы со всех сторон.
- **MaxHeight** (`System.Double`). Получает или задает максимально допустимую высоту элемента в аппаратно-независимых единицах. Данное значение должно быть больше или равно 0.0. Значение по умолчанию — `System.Double.PositiveInfinity`.
- **MaxWidth** (`System.Double`). Получает или задает максимально допустимую ширину элемента в аппаратно-независимых единицах. Данное значение должно быть больше или равно 0.0. Значение по умолчанию — `System.Double.PositiveInfinity`.
- **MinHeight** (`System.Double`). Получает или задает минимально допустимую высоту элемента в аппаратно-независимых единицах. Данное значение должно быть больше или равно 0.0, но не может быть равно `System.Double.PositiveInfinity` или `System.Double.NaN`. Значение по умолчанию — 0.0.
- **MinWidth** (`System.Double`). Получает или задает минимально допустимую ширину элемента в аппаратно-независимых единицах. Данное значение должно быть больше или равно 0.0, но не может быть равно `System.Double.PositiveInfinity` или `System.Double.NaN`. Значение по умолчанию — 0.0.
- **VerticalAlignment** (`System.Windows.VerticalAlignment`). Получает или задает вертикальное выравнивание элемента для тех случаев, когда он помещается внутрь

другого элемента (например, панели). Значение по умолчанию — System.Windows.VerticalAlignment.Stretch.

- **Width** (System.Double). Получает или задает предложенную ширину элемента в аппаратно-независимых единицах. Данное значение должно быть больше или равно 0.0. Значение по умолчанию — System.Double.NaN.

Свойства класса System.Windows.Controls.Control, связанные с позиционированием:

- **HorizontalContentAlignment** (System.Windows.Horizontal Alignment). Получает или задает горизонтальное выравнивание содержимого элемента. Значение по умолчанию — System.Windows.HorizontalAlignment.Left.
- **Padding** (System.Windows.Thickness). Получает или задает внутренние отступы для элемента (между рамкой и содержимым). Значение по умолчанию — объект System.Windows.Thickness, описывающий нулевые отступы со всех сторон.
- **VerticalContentAlignment** (System.Windows.Vertical Alignment). Получает или задает вертикальное выравнивание содержимого элемента. Значение по умолчанию — System.Windows.VerticalAlignment.Top.

12.1. Отступы

Свойство System.Windows.FrameworkElement.Margin отвечает за то, какие отступы должны быть с каждой стороны элемента. Под отступом понимается свободное пространство между внешней границей элемента и внутренней границей его родительского элемента (например, панели). Это свойство задается в аппаратно-независимых

единицах и если его не задавать, то отступы будут нулевые, т.е. будут отсутствовать. У каждой из четырех сторон элемента отступ может быть задан отдельно и отличаться от других. Для описания отступов используется структура System.Windows.Thickness, которая позволяет описать толщину рамки вокруг прямоугольной зоны.

Конструкторы структуры System.Windows.Thickness:

- Thickness(uniformLength). Инициализирует экземпляр структуры, используя указанное значение для всех сторон.
- Thickness(left, top, right, bottom). Инициализирует экземпляр указанными значениями для всех сторон.
- Свойства структуры System.Windows.Thickness:
- Bottom (System.Double). Получает или задает высоту нижней части рамки в аппаратно-независимых единицах. Значение по умолчанию — 0.0.
- Left (System.Double). Получает или задает ширину левой части рамки в аппаратно-независимых единицах. Значение по умолчанию — 0.0.
- Right (System.Double). Получает или задает ширину правой части рамки в аппаратно-независимых единицах. Значение по умолчанию — 0.0.
- Top (System.Double). Получает или задает высоту верхней части рамки в аппаратно-независимых единицах. Значение по умолчанию — 0.0.

Если два элемента описывают отступы по отношению друг к другу, то эти отступы не перекрываются, а суммируются. Например, есть две кнопки, расположенные горизонтально.

зонтально друг за другом. Первая указывает отступ справа 10 аппаратно-независимых единиц, и вторая слева тоже 10 аппаратно-независимых единиц. Это означает, что между кнопками будет отступ 20 аппаратно-независимых единиц.

Структура System.Windows.Thickness имеет преобразователь типа, который позволяет использовать строки определенного формата, для инициализации свойств этого типа. Есть несколько вариантов указания отступов в разметке.

Если указать только одно число, то оно будет использовано для всех сторон. Пример ниже демонстрирует создание кнопки с отступами со всех сторон по 20 аппаратно-независимых единиц.

XAML

```
<Button Margin="20">OK</Button>
```

Если указать два числа через запятую, то первое будет использовано для левой и правой стороны, а второе для верхней и нижней.

XAML

```
<Button Margin="20,50">OK</Button>
```

Если указать четыре числа через запятую, то они будут использованы для каждой отдельной стороны в порядке: левая, верхняя, правая, нижняя.

XAML

```
<Button Margin="20,50,30,40">OK</Button>
```

В качестве примера давайте рассмотрим следующий фрагмент разметки (полный пример находится в Wpf.Properties.Margin.Xaml):

XAML

```
<Grid>
    <Button Margin="10,20,30,40">OK</Button>
</Grid>
```

Получившийся результат демонстрирует кнопку, у которой установлены следующие отступы: слева — 10.0, сверху — 20.0, справа — 30.0, снизу — 40.0 (рис. 31).

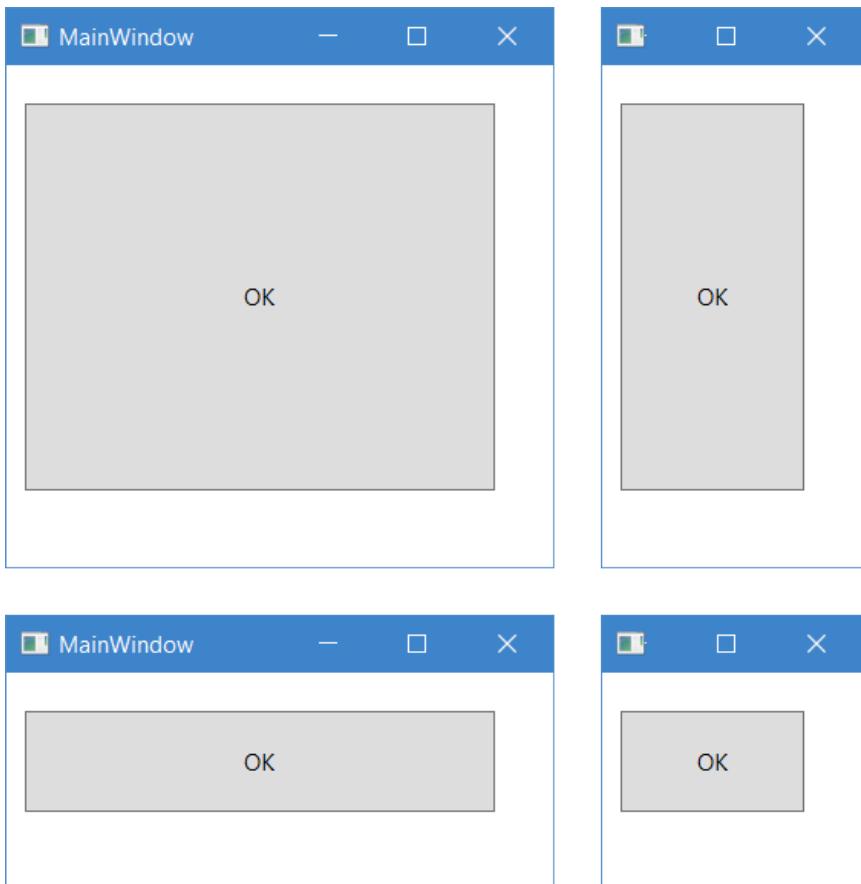


Рис. 31. Использование свойства Margin

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Properties.Margin.CSharp):

C#

```
var button = new Button { Content = "OK",
    Margin = new Thickness(10.0, 20.0, 30.0, 40.0) };

var grid = new Grid();
grid.Children.Add(button);
```

12.2. Выравнивания

Довольно часто возникает ситуация, что доступное пространство для элемента больше, чем ему необходимо. В таких ситуациях используются свойства выравнивания, чтобы вычислить куда именно поместить элемент или растянуть его, чтобы занять все доступное пространство. За это отвечают два свойства: `System.Windows.FrameworkElement.HorizontalAlignment` (горизонтальное выравнивание) и `System.Windows.FrameworkElement.VerticalAlignment` (вертикальное выравнивание).

Для того, чтобы указать горизонтальное выравнивание необходимо использовать перечисление `System.Windows.HorizontalAlignment`, которое содержит следующие варианты:

- Center. Выравнивание по центру.
- Left. Выравнивание по левому краю.
- Right. Выравнивание по правому краю.
- Stretch. Элемент должен растянуться на всю предоставленную ширину.

В качестве примера давайте рассмотрим следующий фрагмент разметки (полный пример находится в Wpf.Properties.HorizontalAlignment.Xaml):

XAML

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

    <Button Grid.Column="0"
            Grid.Row="0" HorizontalAlignment="Left">
        Left
    </Button>
    <Button Grid.Column="0"
            Grid.Row="1" HorizontalAlignment="Center">
        Center
    </Button>
    <Button Grid.Column="0"
            Grid.Row="2" HorizontalAlignment="Right">
        Right
    </Button>
    <Button Grid.Column="0"
            Grid.Row="3" HorizontalAlignment="Stretch">
        Stretch</Button>
</Grid>
```

Получившийся результат демонстрирует создание четырех кнопок (рис. 32). В случае использования любого варианта выравнивания, кроме System.Windows.HorizontalAlignment.Stretch, элемент в ширину будет подгоняться по содержимому.

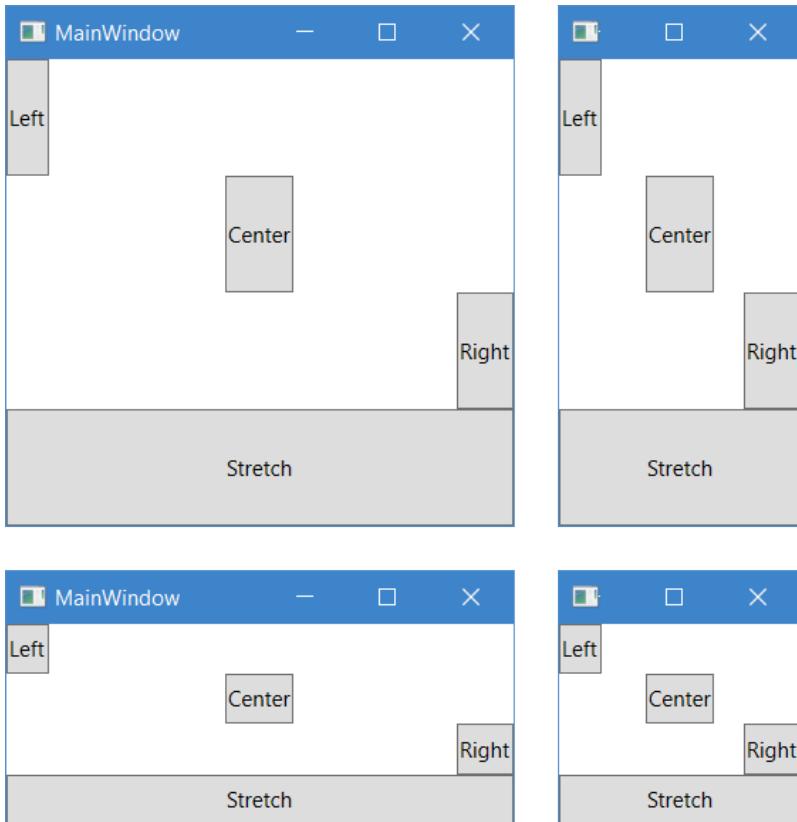


Рис. 32. Использование свойства `HorizontalAlignment`

Рассмотренной выше разметке соответствует следующий код (полный пример находится в `Wpf.Properties.HorizontalAlignment.CSharp`):

C#

```
var button1 = new Button
{
    Content = "Left",
    HorizontalAlignment = HorizontalAlignment.Left
};
```

```
var button2 = new Button
{
    Content = "Center",
    HorizontalAlignment = HorizontalAlignment.Center
};

var button3 = new Button
{
    Content = "Right",
    HorizontalAlignment = HorizontalAlignment.Right
};

var button4 = new Button
{
    Content = "Stretch",
    HorizontalAlignment = HorizontalAlignment.Stretch
};

var grid = new Grid();
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());

grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);

Grid.SetRow(button1, 0);
Grid.SetRow(button2, 1);
Grid.SetRow(button3, 2);
Grid.SetRow(button4, 3);
```

Для того, чтобы указать вертикальное выравнивание необходимо использовать перечисление System.

Windows. **VerticalAlignment**, которое содержит следующие варианты:

- Bottom. Выравнивание по нижнему краю.
- Center. Выравнивание по центру.
- Stretch. Элемент должен растянуться на всю предоставленную высоту.
- Top. Выравнивание по верхнему краю.

В качестве примера давайте рассмотрим следующий фрагмент разметки (полный пример находится в Wpf.Properties.VerticalAlignment.Xaml):

XAML

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

    <Button Grid.Column="0" Grid.Row="0"
           VerticalAlignment="Top">Top
    </Button>
    <Button Grid.Column="1" Grid.Row="0"
           VerticalAlignment="Center">Center
    </Button>
    <Button Grid.Column="2" Grid.Row="0"
           VerticalAlignment="Bottom">Bottom
    </Button>
    <Button Grid.Column="3" Grid.Row="0"
           VerticalAlignment="Stretch">Stretch
    </Button>
</Grid>
```

Получившийся результат демонстрирует создание четырех кнопок (рис. 33). В случае использования любого варианта выравнивания, кроме System.Windows.VerticalAlignment.Stretch, элемент в высоту будет подготавливаться по содержимому.

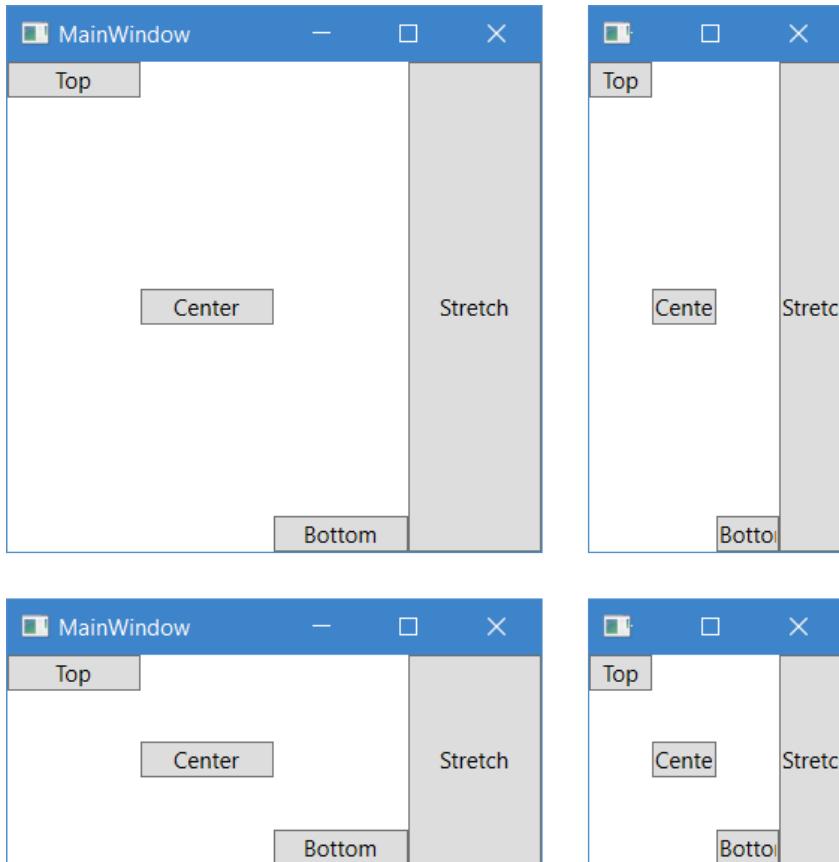


Рис. 33. Использование свойства VerticalAlignment

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Properties.VerticalAlignment.CSharp):

C#

```
var button1 = new Button
{
    Content = "Top",
    VerticalAlignment = VerticalAlignment.Top
};
var button2 = new Button
{
    Content = "Center",
    VerticalAlignment = VerticalAlignment.Center
};
var button3 = new Button
{
    Content = "Bottom",
    VerticalAlignment = VerticalAlignment.Bottom
};
var button4 = new Button
{
    Content = "Stretch",
    VerticalAlignment = VerticalAlignment.Stretch
};

var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.Children.Add(button1);
grid.Children.Add(button2);
grid.Children.Add(button3);
grid.Children.Add(button4);

Grid.SetColumn(button1, 0);
Grid.SetColumn(button2, 1);
Grid.SetColumn(button3, 2);
Grid.SetColumn(button4, 3);
```

12.3. Размеры

Иногда, размеры, предлагаемые элементу в результате расчета компоновки, могут быть не самыми оптимальными, с точки зрения его внешнего вида. Например, довольно распространенная практика, задавать фиксированные размеры кнопкам. Если позволить кнопкам растягиваться на все доступное пространство они часто выглядят нелепо, а если задать им определение размера по содержимому, то они могут выглядеть слишком маленькими.

Несмотря на то, что фиксированные размеры могут быть уместны для некоторых элементов, использование такого подхода не является гибким и поэтому не должно быть вариантом по умолчанию при проектировании интерфейса. Существует более гибкий подход к указанию размеров элемента: использование свойств System.Windows.FrameworkElement.MinWidth и System.Windows.FrameworkElement.MaxWidth для указания минимальной и максимальной ширины, а также System.Windows.FrameworkElement.MinHeight и System.Windows.FrameworkElement.MaxHeight для указания минимальной и максимальной высоты. Эти свойства позволяют задать допустимый диапазон для размеров элемента.

В качестве примера давайте рассмотрим следующий фрагмент разметки (полный пример находится в Wpf.Properties.Width.Xaml):

XAML

```
<StackPanel>
    <Button Margin="5,5,5,0">1</Button>
    <Button Margin="5,5,5,0" Width="100">2</Button>
```

```

<Button Margin="5,5,5,0" MinWidth="100">3</Button>
<Button Margin="5,5,5,0" MaxWidth="100">4</Button>
<Button Margin="5,5,5,0" MinWidth="100"
        Width="150">5</Button>
<Button Margin="5,5,5,0" MinWidth="150"
        Width="100">6</Button>
<Button Margin="5,5,5,0" MaxWidth="100"
        Width="150">7</Button>
<Button Margin="5,5,5,0" MaxWidth="150"
        Width="100">8</Button>
<Button Margin="5,5,5,0" MaxWidth="200"
        MinWidth="150">9</Button>
</StackPanel>

```

Получившийся результат демонстрирует создание кнопок с разными комбинациями указания ширины (рис. 34). Если указать значение свойства `System.Windows.FrameworkElement.Width` противоречащим значениям

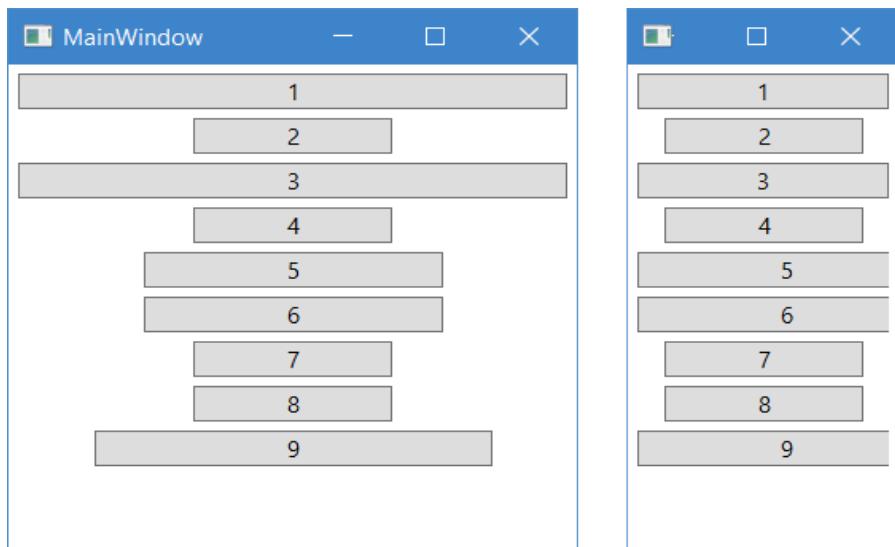


Рис. 34. Использование свойств `Width`, `MinWidth` и `MaxWidth`

свойств `System.Windows.FrameworkElement.MinWidth` или `System.Windows.FrameworkElement.MaxWidth`, то оно будет проигнорировано. Если же задать свойство корректно, но при этом элемент не будет помещаться в предоставленное ему пространство, то он будет усечен.

Рассмотренной выше разметке соответствует следующий код (полный пример находится в `Wpf.Properties.Width.CSharp`):

C#

```
var margin = new Thickness(5.0, 5.0, 5.0, 0.0);

var button1 = new Button {
    Content = 1,
    Margin = margin
};

var button2 = new Button {
    Content = 2,
    Margin = margin,
    Width = 100.0
};

var button3 = new Button {
    Content = 3,
    Margin = margin,
    MinWidth = 100.0
};

var button4 = new Button {
    Content = 4,
    Margin = margin,
    MaxWidth = 100.0
};

var button5 = new Button {
    Content = 5,
    Margin = margin,
```

```
    MinWidth = 100.0,
    Width = 150.0
};

var button6 = new Button {
    Content = 6,
    Margin = margin,
    MinWidth = 150.0,
    Width = 100.0
};

var button7 = new Button {
    Content = 7,
    Margin = margin,
    MaxWidth = 100.0,
    Width = 150.0
};

var button8 = new Button {
    Content = 8,
    Margin = margin,
    MaxWidth = 150.0,
    Width = 100.0
};

var button9 = new Button{
    Content = 9,
    Margin = margin,
    MaxWidth = 200.0,
    MinWidth = 150.0
};

var stackPanel = new StackPanel();
stackPanel.Children.Add(button1);
stackPanel.Children.Add(button2);
stackPanel.Children.Add(button3);
stackPanel.Children.Add(button4);
stackPanel.Children.Add(button5);
stackPanel.Children.Add(button6);
stackPanel.Children.Add(button7);
stackPanel.Children.Add(button8);
stackPanel.Children.Add(button9);
```

Свойства, связанные с высотой элемента, работают точно также, как и свойства, связанные с шириной, рассмотренные выше.

В качестве примера давайте рассмотрим следующий фрагмент разметки (полный пример находится в Wpf.Properties.Height.Xaml):

XAML

```
<StackPanel Orientation="Horizontal">
    <Button Margin="5,5,0,5">1</Button>
    <Button Height="100" Margin="5,5,0,5">2</Button>
    <Button Margin="5,5,0,5" MinHeight="100">3
        </Button>
    <Button Margin="5,5,0,5" MaxHeight="100">4
        </Button>
    <Button Height="150" Margin="5,5,0,5"
        MinHeight="100">5</Button>
    <Button Height="100" Margin="5,5,0,5"
        MinHeight="150">6</Button>
    <Button Height="150" Margin="5,5,0,5"
        MaxHeight="100">7</Button>
    <Button Height="100" Margin="5,5,0,5"
        MaxHeight="150">8</Button>
    <Button Margin="5,5,0,5" MaxHeight="200"
        MinHeight="150">9</Button>
</StackPanel>
```

Получившийся результат демонстрирует создание кнопок с разными комбинациями указания высоты (рис. 35). Если указать значение свойства System.Windows.FrameworkElement.Height противоречащим значениям свойств System.Windows.FrameworkElement.MinHeight или System.Windows.FrameworkElement.MaxHeight, то оно будет проигнорировано. Если же задать свойство

корректно, но при этом элемент не будет помещаться в предоставленное ему пространство, то он будет усечен.

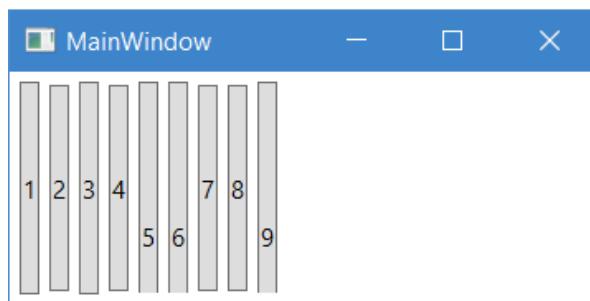
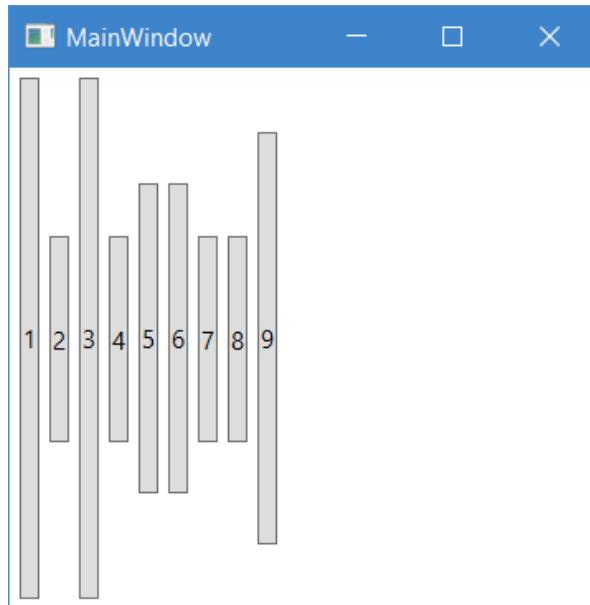


Рис. 35. Использование свойств Height, MinHeight и MaxHeight

Рассмотренной выше разметке соответствует следующий код (полный пример находится в `Wpf.Properties.Height.CSharp`):

C#

```
var margin = new Thickness(5.0, 5.0, 0.0, 5.0);

var button1 = new Button {
    Content = 1,
    Margin = margin
};

var button2 = new Button {
    Content = 2,
    Height = 100.0,
    Margin = margin
};

var button3 = new Button {
    Content = 3,
    Margin = margin,
    MinHeight = 100.0
};

var button4 = new Button {
    Content = 4,
    Margin = margin,
    MaxHeight = 100.0
};

var button5 = new Button {
    Content = 5,
    Height = 150.0,
    Margin = margin,
    MinHeight = 100.0
};

var button6 = new Button {
    Content = 6,
    Height = 100.0,
    Margin = margin,
    MinHeight = 150.0
};

var button7 = new Button {
    Content = 7,
```

```
    Height = 150.0,  
    Margin = margin,  
    MaxHeight = 100.0  
};  
var button8 = new Button {  
    Content = 8,  
    Height = 100.0,  
    Margin = margin,  
    MaxHeight = 150.0  
};  
var button9 = new Button {  
    Content = 9,  
    Margin = margin,  
    MaxHeight = 200.0,  
    MinHeight = 150.0  
};  
  
var stackPanel = new StackPanel { Orientation =  
    Orientation.Horizontal };  
  
stackPanel.Children.Add(button1);  
stackPanel.Children.Add(button2);  
stackPanel.Children.Add(button3);  
stackPanel.Children.Add(button4);  
stackPanel.Children.Add(button5);  
stackPanel.Children.Add(button6);  
stackPanel.Children.Add(button7);  
stackPanel.Children.Add(button8);  
stackPanel.Children.Add(button9);
```

Явное указание значений для свойств System.Windows.FrameworkElement.Width и System.Windows.FrameworkElement.Height имеет больший приоритет, чем требование того, чтобы элемент был растянут. Другими словами, если задать свойство System.Windows.FrameworkElement.Width и установить значение свойству System.Windows.

`FrameworkElement.HorizontalAlignment` равным `System.Windows.HorizontalAlignment.Stretch`, то последнее будет проигнорировано. Аналогичная ситуация происходит со свойствами `System.Windows.FrameworkElement.Height` и `System.Windows.FrameworkElement.VerticalAlignment`.

В определенных ситуациях может понадобиться узнать реальные размеры элемента. В этом случае не стоит использовать свойства `System.Windows.FrameworkElement.Width` и `System.Windows.FrameworkElement.Height`, так как они отражают желаемые размеры, а не действительные. Стоит использовать свойства `System.Windows.FrameworkElement.ActualWidth` и `System.Windows.FrameworkElement.ActualHeight`, которые хранят размеры, используемые при визуализации элемента.

13. Кнопки

Одним из наиболее часто встречающихся элементов управления является кнопка. В WPF присутствует несколько видов кнопок, которые будут рассмотрены в этом разделе.

- `System.Windows.Controls.Button`. Обычная кнопка.
- `System.Windows.Controls.CheckBox`. Флажок, который позволяет управлять двумя состояниями: включено и выключено.
- `System.Windows.Controls.RadioButton`. Переключатель, который позволяет выбрать одну опцию из предоставленного набора (группы).

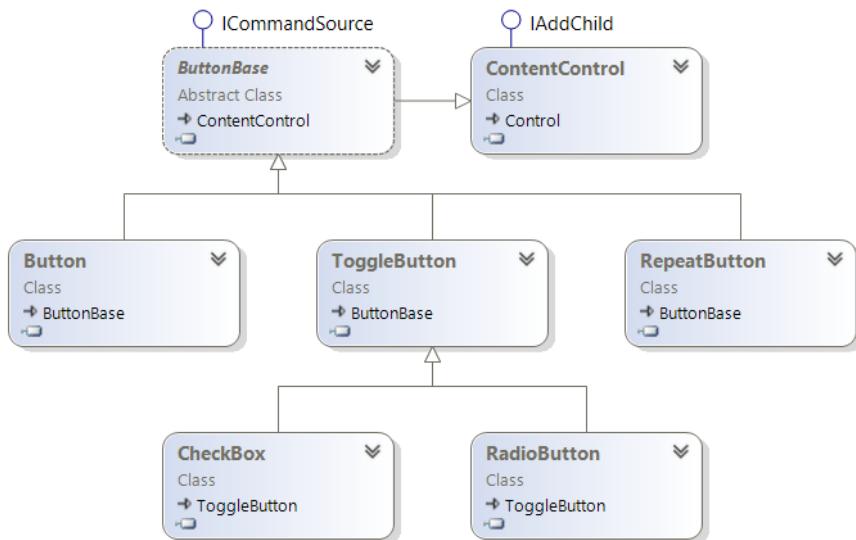


Рис. 36. Диаграмма классов, описывающих кнопки

- System.Windows.Controls.Primitives.RepeatButton. Кнопка, которая постоянно генерирует событие щелчка, с момента нажатия и до тех пор, пока не будет отпущена.

Несмотря на то, что это разные элементы управления, описанные при помощи отдельных классов, они все имеют одну и ту же базовую функциональность, описанную в базовых классах. На рис. 36 показана диаграмма классов, описывающих кнопки.

На диаграмме выше присутствует несколько базовых классов, от которых наследуются рассмотренные выше элементы управления:

- System.Windows.Controls.Primitives.ButtonBase. Данный класс является базовым для всех кнопок. Здесь собрана все общая функциональность присущая кнопкам любого вида.
- System.Windows.Controls.Primitives.ToggleButton. Данный класс является базовым для кнопок, которые поддерживают два состояния: включено и выключено.

Свойства класса System.Windows.Controls.Primitives.ButtonsBase:

- ClickMode (System.Windows.Controls.ClickMode). Получает или задает режим срабатывания события щелчка кнопки. Значение по умолчанию — System.Windows.Controls.ClickMode.Release.
- IsPressed (System.Boolean). Получает значение, которое указывает нажата ли кнопка. **true** если кнопка нажата; иначе — **false**.

События класса System.Windows.Controls.Primitives.
ButtonBase:

- **Click** (System.Windows.RoutedEventHandler). Срабатывает, когда кнопка нажимается.

Для того, чтобы указать режим срабатывания события щелчка кнопки необходимо использовать перечисление System.Windows.Controls.**ClickMode**, которое содержит следующие варианты:

- Hover. Событие щелчка должно сработать при наведении курсора мыши на кнопку.
- Press. Событие щелчка должно сработать при нажатии кнопки.
- Release. Событие щелчка должно сработать при отпускании кнопки.

Свойства класса System.Windows.Controls.Primitives.
ToggleButton:

- **IsChecked** (System.Nullable<System.Boolean>). Получает или задает значение, указывающее включена кнопка или нет. **True** если включена; **false** если кнопка выключена; иначе — **null**. Значение по умолчанию — **false**.
- **IsThreeState** (System.Windows.Boolean). Получает или задает значение, которое указывает на то, поддерживает ли кнопка три состояния переключения. **true** если кнопка поддерживает три состояния; иначе — **false**. Значение по умолчанию — **false**.

События класса System.Windows.Controls.Primitives.
ToggleButton:

- **Checked** (System.Windows.RoutedEventHandler). Срабатывает, когда кнопка включается.

- **Indeterminate** (System.Windows.RoutedEventHandler). Срабатывает, когда кнопка переводится в третье состояние (не включена и не выключена).
- **Unchecked** (System.Windows.RoutedEventHandler). Срабатывает, когда кнопка выключается.

13.1. Элемент управления Button

Элемент управления System.Windows.Controls.Button является самым распространенным видом кнопки, использующимся при проектировании пользовательского интерфейса. Главная задача данного элемента управления обрабатывать нажатие и генерировать при этом соответствующее событие, с которым, как правило, связано определенное программное действие.

Приведенный ниже фрагмент разметки демонстрирует элемент управления (полный пример находится в папке Wpf.Controls.Button.Xaml):

XAML

```
<Button Click="Button_Click" Height="23"  
Width="75">Button</Button>
```

Результат приведенной выше разметки показан на рис. 37.

В данном примере показано, как подписатьсь на событие кнопки System.Windows.Controls.ButtonBase.Click, которое отвечает за ее нажатие. В качестве названия атрибута необходимо написать название события, а в качестве значения — название метода-обработчика события. Это означает, что в файле с кодом, который соответствует текущему файлу

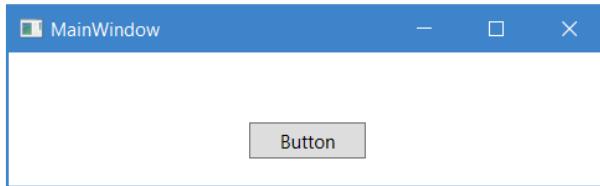


Рис. 37. Элемент управления Button

разметки (*code-behind*) должен быть метод, с таким называнием и сигнатурой подходящей делегату события:

C#

```
private void Button_Click(object sender,
                         RoutedEventArgs e)
{
    Title = $"Button clicked at {DateTime.Now}";
}
```

Рассмотренной выше разметке соответствует следующий код (полный пример находится в `Wpf.Controls.Button.CSharp`):

C#

```
var button = new Button {
    Content = "Button",
    Height = 23.0,
    Width = 75.0 };
button.Click += Button_Click;
```

13.2. Элемент управления RepeatButton

Элемент управления `System.Windows.Controls.Primitives.RepeatButton` отличается от обычной кнопки только тем, что он генерирует событие щелчка после нажатия и до того, как кнопки будет отпущена.

Свойства класса System.Windows.Controls.Primitives.RepeatButton:

- **Delay** (System.Int32). Получает или задает количество миллисекунд, которое должно пройти перед тем, как начать генерировать события щелчка, после того, как кнопка была нажата. Значение по умолчанию — System.Windows.SystemParameters.KeyboardDelay.
- **Interval** (System.Int32). Получает или задает количество миллисекунд, которое должно пройти перед повторным событием щелчка. Значение по умолчанию — System.Windows.SystemParameters.KeyboardSpeed.

Приведенный ниже фрагмент разметки демонстрирует элемент управления (полный пример находится в папке Wpf.Controls.RepeatButton.Xaml):

XAML

```
<RepeatButton Click="RepeatButton_Click"
               Height="23" Width="100"> Repeat Button
</RepeatButton>
```

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Controls.RepeatButton.CSharp):

C#

```
var repeatButton = new RepeatButton
{
    Content = "Repeat Button",
    Height = 23.0,
    Width = 100.0
};
repeatButton.Click += RepeatButton_Click;
```

13.3. Элемент управления CheckBox

Элемент управления System.Windows.Controls.**CheckBox** позволяет пользователю оперировать неким параметром с двумя состояниями: включено и выключено. При необходимости возможно создать элемент, который будет поддерживать не два, а три состояния. Помимо того, что элемент генерирует событие при каждом нажатии, он также генерирует более детальные события, связанные с каждым из состояний (включение, выключение).

Приведенный ниже фрагмент разметки демонстрирует элемент управления (полный пример находится в папке Wpf.Controls.CheckBox.Xaml):

XAML

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

    <CheckBox Checked="CheckBox_Checked"
        Grid.Row="0"
        HorizontalAlignment="Center"
        Unchecked="CheckBox_Unchecked"
        VerticalAlignment="Center">
        Unchecked
    </CheckBox>

    <CheckBox Checked="CheckBox_Checked"
        Grid.Row="1"
        HorizontalAlignment="Center"
        IsChecked="True"
        VerticalAlignment="Center">
        Checked
    </CheckBox>
```

```

        Unchecked="CheckBox_Unchecked"
        VerticalAlignment="Center">
    Checked
</CheckBox>

<CheckBox Checked="CheckBox_Checked"
    Grid.Row="2"
    HorizontalAlignment="Center"
    Indeterminate="CheckBox_Indeterminate"
    IsChecked="{x:Null}"
    IsThreeState="True"
    Unchecked="CheckBox_Unchecked"
    VerticalAlignment="Center">
    Indeterminate
</CheckBox>
</Grid>

```

Результат приведенной выше разметки показан на рис. 38.



Рис. 38. Элемент управления CheckBox

В данном примере использовано расширение разметки для того, чтобы присвоить значение **null** одному из свойств: **IsThreeState="{}x:Null{}"**.

Рассмотренной выше разметке соответствует следующий код (полный пример находится в **Wpf.Controls.CheckBox.CSharp**):

C#

```
var checkBox1 = new CheckBox
{
    Content = "Unchecked",
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
checkBox1.Checked += CheckBox_Checked;
checkBox1.Unchecked += CheckBox_Unchecked;

var checkBox2 = new CheckBox
{
    Content = "Checked",
    HorizontalAlignment = HorizontalAlignment.Center,
    IsChecked = true,
    VerticalAlignment = VerticalAlignment.Center
};
checkBox2.Checked += CheckBox_Checked;
checkBox2.Unchecked += CheckBox_Unchecked;

var checkBox3 = new CheckBox
{
    Content = "Indeterminate",
    HorizontalAlignment = HorizontalAlignment.Center,
    IsChecked = null,
    IsThreeState = true,
    VerticalAlignment = VerticalAlignment.Center
};
checkBox3.Checked += CheckBox_Checked;
checkBox3.Indeterminate += CheckBox_Indeterminate;
checkBox3.Unchecked += CheckBox_Unchecked;

var grid = new Grid();
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
```

```
grid.Children.Add(checkBox1);
grid.Children.Add(checkBox2);
grid.Children.Add(checkBox3);
Grid.SetRow(checkBox1, 0);
Grid.SetRow(checkBox2, 1);
Grid.SetRow(checkBox3, 2);
```

Элемент управления RadioButton

Элемент управления System.Windows.Controls. **RadioButton** позволяет пользователю выбрать одну опцию (пункт) из доступного набора (группы). При активации одного элемента из группы, все остальные элементы этой группы автоматически деактивируются, предоставляя функциональность выбора только одного из элементов. При этом в пределах интерфейса может быть несколько несвязанных между собой групп переключателей.

Свойства класса System.Windows.Controls. **RadioButton**:

- **GroupName** (System.String). Получает или задает название группы, к которой принадлежит переключатель. Значение по умолчанию — System.String.Empty.

Приведенный ниже фрагмент разметки демонстрирует элемент управления (полный пример находится в папке Wpf.Controls.RadioButton.Xaml):

XAML

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

```
<StackPanel Grid.Column="0"
    HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <RadioButton Click="RadioButton_Click"
        Margin="5">Option 1</RadioButton>
    <RadioButton Click="RadioButton_Click"
        Margin="5">Option 2</RadioButton>
    <RadioButton Click="RadioButton_Click"
        Margin="5">Option 3</RadioButton>
</StackPanel>

<StackPanel Grid.Column="1"
    HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <RadioButton Click="RadioButton_Click"
        Margin="5">Option 4</RadioButton>
    <RadioButton Click="RadioButton_Click"
        Margin="5">Option 5</RadioButton>
    <RadioButton Click="RadioButton_Click"
        Margin="5">Option 6</RadioButton>
</StackPanel>

</Grid>
```

Результат приведенной выше разметки показан на рис. 39.

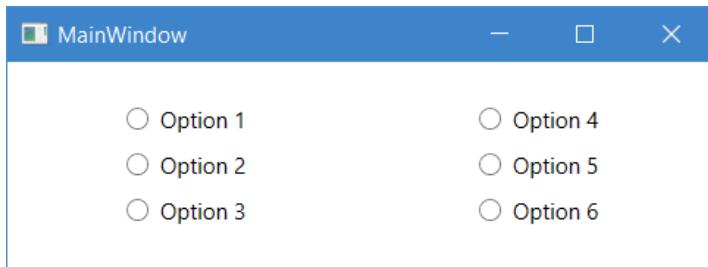


Рис. 39. Элемент управления RadioButton.

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Controls.RadioButton.CSharp):

C#

```
var margin = new Thickness(5.0);

var radioButton1 = new RadioButton {
    Content = "Option 1",
    Margin = margin
};

radioButton1.Click += RadioButton_Click;
var radioButton2 = new RadioButton {
    Content = "Option 2",
    Margin = margin
};

radioButton2.Click += RadioButton_Click;
var radioButton3 = new RadioButton {
    Content = "Option 3",
    Margin = margin
};

radioButton3.Click += RadioButton_Click;
var radioButton4 = new RadioButton {
    Content = "Option 4",
    Margin = margin
};

radioButton4.Click += RadioButton_Click;
var radioButton5 = new RadioButton {
    Content = "Option 5",
    Margin = margin
};

radioButton5.Click += RadioButton_Click;
var radioButton6 = new RadioButton {
    Content = "Option 6",
    Margin = margin
};

radioButton6.Click += RadioButton_Click;
```

```
var stackPanel1 = new StackPanel
{
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
stackPanel1.Children.Add radioButton1;
stackPanel1.Children.Add radioButton2;
stackPanel1.Children.Add radioButton3;

var stackPanel2 = new StackPanel
{
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
stackPanel2.Children.Add radioButton4;
stackPanel2.Children.Add radioButton5;
stackPanel2.Children.Add radioButton6;

var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.Children.Add(stackPanel1);
grid.Children.Add(stackPanel2);

Grid.SetColumn(stackPanel1, 0);
Grid.SetColumn(stackPanel2, 1);
```

Группировать переключатели можно двумя способами:

- Все переключатели, находящиеся внутри одной панели или группирующего элемента управления, автоматически помещаются в одну группу.
- Все переключатели, с одинаковым значением свойства System.Windows.Controls.RadioButton.GroupName, автоматически помещаются в одну группу. Этот ва-

риант группировки не учитывает местонахождение элемента в дереве элементов. Другими словами, при помощи этого способа можно создать более одной группы в пределах одной панели или создать одну группу, элементы которой будут находиться в разных панелях.

14. Поля для ввода текста

Наиболее распространенным элементом управления для получения данных от пользователя является поле для ввода текста. В WPF присутствует несколько видов таких полей, которые будут рассмотрены в этом разделе.

- System.Windows.Controls.TextBox. Обычное поле для ввода текста.
- System.Windows.Controls.RichTextBox. Поле для ввода текста, которое поддерживает разнообразное форматирование текста: выравнивание, шрифт, цвет и т.д. Данный элемент будет рассмотрен в разделе документов

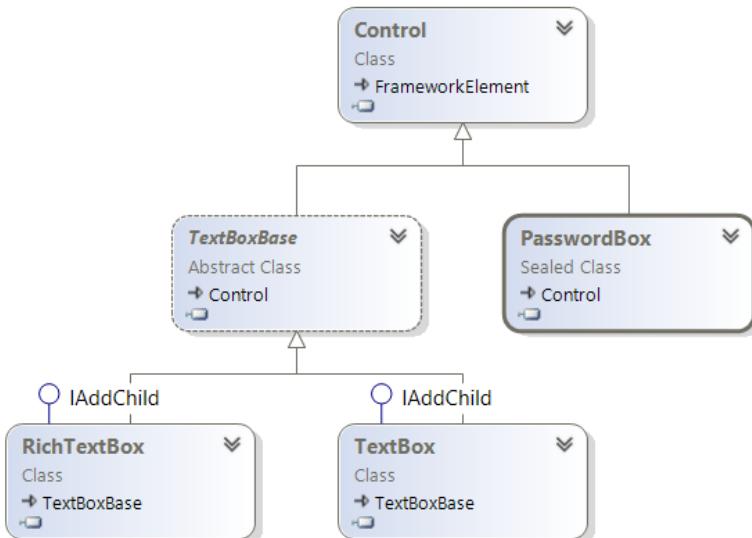


Рис. 40. Диаграмма классов, описывающих поля для ввода текста

нефиксированного формата, так как он применяет их для отображения своего содержимого.

- System.Windows.Controls.PasswordBox. Текстовое поле для ввода и обработки паролей.

У разных текстовых полей, как и у кнопок, есть часть общей функциональности, которая вынесена в базовые классы. На рис. 40 показана диаграмма классов, описывающих поля для ввода текста.

На диаграмме выше присутствует базовый класс System.Windows.Controls.Primitives.TextBoxBase, от которого наследуются классы System.Windows.Controls.TextBox и System.Windows.Controls.RichTextBox.

Свойства класса System.Windows.Primitives.TextBoxBase:

- AcceptsReturn (System.Boolean). Получает или задает значение, которое указывает каким образом элемент реагирует на нажатие клавиши *Enter*. **true** если при нажатии клавиши *Enter* необходимо вставить переход на новую строку в текущую позицию курсора ввода; иначе — нажатие клавиши *Enter* игнорируется. Значение по умолчанию — **false** для класса System.Windows.Controls.TextBox и **true** для класса System.Windows.Controls.RichTextBox.
- AcceptsTab (System.Boolean). Получает или задает значение, которое указывает каким образом элемент реагирует на нажатие клавиши *Tab*. **true** если при нажатии клавиши *Tab* необходимо вставить символ горизонтальной табуляции в текущую позицию курсора ввода; иначе — необходимо переместить фокус на следующий элемент и не вставлять символ горизонтальной табуляции. Значение по умолчанию — **false**.

- **AutoWordSelection** (System.Boolean). Получает или задает значение, которое указывает необходимо ли выделять слово целиком, когда пользователь выделяет часть слова и перемещает курсор мыши вдоль него. **true** если автоматическое выделение слова включено; иначе — **false**. Значение по умолчанию — **false**.
- **CanRedo** (System.Boolean). Получает значение, которое указывает можно ли повторить последнее отмененное действие. **true** если можно; иначе — **false**.
- **CanUndo** (System.Boolean). Получает значение, которое указывает можно ли отменить последнее действие. **true** если можно; иначе — **false**.
- **CaretBrush** (System.Windows.Media.Brush). Получает или задает кисть, которая используется для отображения курсора ввода.
- **HorizontalOffset** (System.Double). Получает значение горизонтальной позиции прокрутки в аппаратно-независимых единицах.
- **HorizontalScrollBarVisibility** (System.Windows.Controls.ScrollBarVisibility). Получает или задает режим отображения горизонтальной полосы прокрутки. Значение по умолчанию — System.Windows.Controls.ScrollBarVisibility.Hidden.
- **IsInactiveSelectionHighlightEnabled** (System.Boolean). Получает или задает значение, которое указывает необходимо ли отображать выделение текста, когда элемент не обладает фокусом. **true** если необходимо отображать выделение; иначе — **false**. Значение по умолчанию — **false**.

- **IsReadOnly** (System.Boolean). Получает или задает значение, которое указывает может ли пользователь редактировать текст в элементе. **true** если не может; иначе — **false**. Значение по умолчанию — **false**.
- **IsReadOnlyCaretVisible** (System.Boolean). Получает или задает значение, которое указывает необходимо ли отображать курсор ввода в элементе, в котором запрещено редактирование текста пользователем. **true** если курсор ввода необходимо отображать; иначе — **false**. Значение по умолчанию — **false**.
- **IsSelectionActive** (System.Boolean). Получает значение, которое указывает обладает ли элемент фокусом и есть ли в нем выделенный текст. **true** если есть фокус и выделение; иначе — **false**.
- **IsUndoEnabled** (System.Boolean). Получает или задает значение, которое указывает необходима ли поддержка отмены последнего действия в элементе. **true** если поддержка отмены последнего действия необходима; иначе — **false**. Значение по умолчанию — **true**.
- **SelectionBrush** (System.Windows.Media.Brush). Получает или задает кисть, которая используется для отображения выделенного текста.
- **SelectionOpacity** (System.Double). Получает или задает степень прозрачности кисти, используемой для отображения выделенного текста. Данное свойство может принимать значения от 0.0 (прозрачный) до 1.0 (непрозрачный). Значение по умолчанию — 0.4.
- **SpellCheck** (System.Windows.Controls.SpellCheck). Получает объект, который предоставляет доступ

к орфографическим ошибкам, допущенным в тексте элемента.

- **UndoLimit** (System.Int32). Получает или задает максимальное количество действий, располагающихся в очереди действий, доступных для отмены. Значение по умолчанию — -1, что означает неограниченный размер очереди.
- **VerticalOffset** (System.Double). Получает значение вертикальной позиции прокрутки в аппаратно-независимых единицах.
- **VerticalScrollBarVisibility** (System.Windows.Controls.ScrollBarVisibility). Получает или задает режим отображения вертикальной полосы прокрутки. По умолчанию — System.Windows.Controls.ScrollBarVisibility.Hidden.

События класса System.Windows.Controls.Primitives.

[TextBoxBase:](#)

- **SelectionChanged** (System.Windows.RoutedEventHandler). Срабатывает, когда изменяется выделение текста.
- **TextChanged** (System.Windows.Controls.TextChangedEventHandler). Срабатывает, когда изменяется текст.
- Методы класса System.Windows.Controls.Primitives.[TextBoxBase](#):
- **AppendText(textData)**. Дописывает переданную строку в конец текста.
- **Copy()**. Копирует текущее выделение текста в буфер обмена.
- **Cut()**. Копирует текущее выделение текста в буфер обмена и удаляет его из текста.

- **LineDown()**. Прокручивает содержимое элемента на одну строку вниз.
- **LineLeft()**. Прокручивает содержимое элемента на одну строку влево.
- **LineRight()**. Прокручивает содержимое элемента на одну строку вправо.
- **LineUp()**. Прокручивает содержимое элемента на одну строку вверх.
- **LockCurrentUndoUnit()**. Завершает самый последний блок отмены действия. Это предотвращает завершенный блок от слияния с последующими блоками действий, которые будут добавлены впоследствии.
- **PageDown()**. Прокручивает содержимое элемента на одну страницу вниз.
- **PageLeft()**. Прокручивает содержимое элемента на одну страницу влево.
- **PageRight()**. Прокручивает содержимое элемента на одну страницу вправо.
- **PageUp()**. Прокручивает содержимое элемента на одну страницу вверх.
- **Paste()**. Вставляет содержимое из буфера обмена вместо текущего выделения текста.
- **Redo()**. Повторяет последнее отмененное действие. Возвращает **true**, если операция выполнилась успешно; иначе — **false** (нет команды для повтора).
- **ScrollToEnd()**. Прокручивает содержимое элемента в конец.

- `ScrollToHome()`. Прокручивает содержимое элемента в начало.
- `ScrollToHorizontalOffset(offset)`. Прокручивает содержимое элемента до переданного значения горизонтальной прокрутки.
- `ScrollToVerticalOffset(offset)`. Прокручивает содержимое элемента до переданного значения вертикальной прокрутки.
- `SelectAll()`. Выделяет весь текст.
- `Undo()`. Отменяет последнее действие. Возвращает `true`, если операция выполнилась успешно; иначе — `false` (нет команды для отмены).

Для того, чтобы указать режим отображения горизонтальной или вертикальной полосы прокрутки необходимо использовать перечисление `System.Windows.Controls.ScrollBarVisibility`, которое содержит следующие варианты:

- `Auto`. Полоса прокрутки появляется автоматически, если содержимое не помещается в клиентскую область элемента.
- `Disabled`. Полоса прокрутки не появляется, даже если содержимое не помещается в клиентскую область элемента. При этом содержимое использует ограниченный размер свободной области.
- `Hidden`. Полоса прокрутки не появляется, даже если содержимое не помещается в клиентскую область элемента. При этом содержимое ведет себя так же, как и при наличии полосы прокрутки.
- `Visible`. Полоса прокрутки видна всегда.

14.1. Элемент управления TextBox

Элемент управления System.Windows.Controls.TextBox является основным видом поля для ввода текста. Поддерживается односторонним и многострочным ввод/вывод текста.

Свойства класса System.Windows.Controls.TextBox:

- **CaretIndex** (System.Int32). Получает или задает позицию курсора ввода. Данное свойство может принимать значения от 0 и больше.
- **CharacterCasing** (System.Windows.Controls.Character Casing). Получает или задает режим изменения регистра введенных пользователем символов. Значение по умолчанию — System.Windows.Controls.Character Casing.Normal.
- **LineCount** (System.Int32). Получает количество строк в тексте.
- **MaxLength** (System.Int32). Получает или задает максимальное количество введенных символов. Значение по умолчанию — 0, что означает неограниченное количество.
- **MaxLines** (System.Int32). Получает или задает максимальное количество видимых строк. Значение по умолчанию — System.Int32.MaxValue.
- **MinLines** (System.Int32). Получает или задает минимальное количество видимых строк. Значение по умолчанию — 1.
- **SelectedText** (System.String). Получает или задает содержимое текущего выделения текста.

- **SelectionLength** (System.Int32). Получает или задает количество символов текущего выделения текста. Значение по умолчанию — 0.
- **SelectionStart** (System.Int32). Получает или задает индекс первого символа текущего выделения текста.
- **Text** (System.String). Получает или задает текст элемента. Значение по умолчанию — System.String.Empty.
- **TextAlignment** (System.Windows.TextAlignment). Получает или задает горизонтальное выравнивание текста. Значение по умолчанию — System.Windows.TextAlignment.Left.
- **TextWrapping** (System.Windows.TextWrapping). Получает или задает режим переноса текста. Значение по умолчанию — System.Windows.TextWrapping.NoWrap.
- Методы класса System.Windows.Controls.TextBox:
 - **Clear()**. Очищает содержимое (текст) элемента.
 - **GetCharacterIndexFromIndex(lineIndex)**. Возвращает индекс первого символа в строке с указанным индексом.
 - **GetFirstVisibleLineIndex()**. Возвращает индекс первой видимой строки.
 - **GetLastVisibleLineIndex()**. Возвращает индекс последней видимой строки.
 - **GetLineIndexFromCharacterIndex(charIndex)**. Возвращает индекс строки, которая содержит символ с указанным индексом.
 - **GetLineLength(lineIndex)**. Возвращает количество символов в строке с указанным индексом.
 - **GetLineText(lineIndex)**. Возвращает содержимое строки с указанным индексом.

- **GetNextSpellingErrorCharacterIndex(charIndex, direction).** Возвращает индекс первого символа следующей орфографической ошибки. Дополнительно указывается индекс символа, с которого необходимо начать поиск и направление поиска.
- **GetSpellingError(charIndex).** Возвращает объект, описывающий орфографическую ошибку по указанному индексу.
- **GetSpellingErrorLength(charIndex).** Возвращает количество символов орфографической ошибки, включаяющей указанный индекс.
- **GetSpellingErrorStart(charIndex).** Возвращает индекс первого символа орфографической ошибки, включаяющей указанный индекс.
- **ScrollToLine(lineIndex).** Прокручивает содержимое элемента до строки с указанным индексом.
- **Select(start, length).** Выделяет текст в указанном диапазоне.

Для того, чтобы указать режим изменения регистра введенных пользователем символов необходимо использовать перечисление `System.Windows.Controls.CharacterCasing`, которое содержит следующие варианты:

- **Lower.** Введенные символы преобразуются в нижний регистр.
- **Normal.** Регистр символов не меняется при вводе.
- **Upper.** Введенные символы преобразуются в верхний регистр.

Для того, чтобы указать горизонтальное выравнивание текста необходимо использовать перечисление `System.`

Windows.Controls.TextAlignment, которое содержит следующие варианты:

- Center. Центрирование текста.
- Justify. Растигивание текста.
- Left. Выравнивание текста по левому краю.
- Right. Выравнивание текста по правому краю.

Для того, чтобы указать режим переноса текста необходимо использовать перечисление System.Windows.TextWrapping, которое содержит следующие варианты:

- NoWrap. Перенос отключен.
- Wrap. Слова переносятся на следующую строку если не помещаются. Если алгоритм переноса не может определить лучший вариант переноса (например, очень длинное слово), перенос все равно осуществляется.
- WrapWithOverflow. Слова переносятся на следующую строку если не помещаются. Если алгоритм переноса не может определить лучший вариант переноса (например, очень длинное слово), перенос не осуществляется, и слово может выйти за границы элемента.

Приведенный ниже фрагмент разметки демонстрирует элемент управления (полный пример находится в папке Wpf.Controls.TextBox.Xaml):

XAML

```
<TextBox HorizontalAlignment="Center"
         Text="Text" VerticalAlignment="Center"
         Width="200"/>
```

Результат приведенной выше разметки показан на рис. 41.

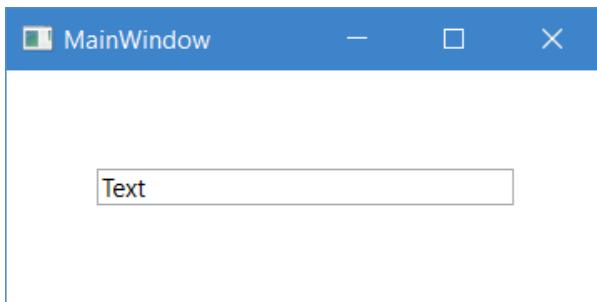


Рис. 41. Элемент управления TextBox

Рассмотренной выше разметке соответствует следующий код (полный пример находится в `Wpf.Controls.TextBox.CSharp`):

C#

```
var textBox = new TextBox
{
    HorizontalAlignment = HorizontalAlignment.Center,
    Text = "Text",
    VerticalAlignment = VerticalAlignment.Center,
    Width = 200.0
};
```

14.2. Элемент управления PasswordBox

Элемент управления `System.Windows.Controls.PasswordBox` предназначен для ввода паролей.

Свойства класса `System.Windows.Controls.PasswordBox`:

- **CaretBrush** (`System.Windows.Media.Brush`). Получает или задает кисть, которая используется для отображения курсора ввода.
- **IsInactiveSelectionHighlightEnabled** (`System.Boolean`). Получает или задает значение, которое указывает

необходимо ли отображать выделение текста, когда элемент не обладает фокусом. **true** если необходимо отображать выделение; иначе — **false**. Значение по умолчанию — **false**.

- **IsSelectionActive** (`System.Boolean`). Получает значение, которое указывает обладает ли элемент фокусом и есть ли в нем выделенный текст. **true** если есть фокус и выделение; иначе — **false**.
 - **MaxLength** (`System.Int32`). Получает или задает максимальное количество введенных символов. Значение по умолчанию — 0, что означает неограниченное количество.
 - **Password** (`System.String`). Получает или задает пароль. Значение по умолчанию — `System.String.Empty`.
 - **PasswordChar** (`System.Char`). Получает или задает символ, который используется подмены и отображения символов пароля. Значение по умолчанию — символ «точка маркер списка» (•).
 - **SecurePassword** (`System.Security.SecureString`). Получает пароль в виде безопасной строки.
 - **SelectionBrush** (`System.Windows.Media.Brush`). Получает или задает кисть, которая используется для отображения выделенного текста.
 - **SelectionOpacity** (`System.Double`). Получает или задает степень прозрачности кисти, используемой для отображения выделенного текста. Данное свойство может принимать значения от 0.0 (прозрачный) до 1.0 (непрозрачный). Значение по умолчанию — 0.4.
- События класса `System.Windows.Controls.PasswordBox`:

- **PasswordChanged** (System.Windows.RoutedEventHandler). Срабатывает, когда изменяется пароль.
- Методы класса System.Windows.Controls.PasswordBox:
- **Clear()**. Очищает содержимое (текст) элемента.
- **Paste()**. Вставляет содержимое из буфера обмена вместо текущего выделения текста.
- **SelectAll()**. Выделяет весь текст.

Приведенный ниже фрагмент разметки демонстрирует элемент управления (полный пример находится в папке Wpf.Controls.PasswordBox.Xaml):

XAML

```
<PasswordBox HorizontalAlignment="Center"
              Password="Text"
              VerticalAlignment="Center"
              Width="200"/>
```

Результат приведенной разметки показан на рис. 42.

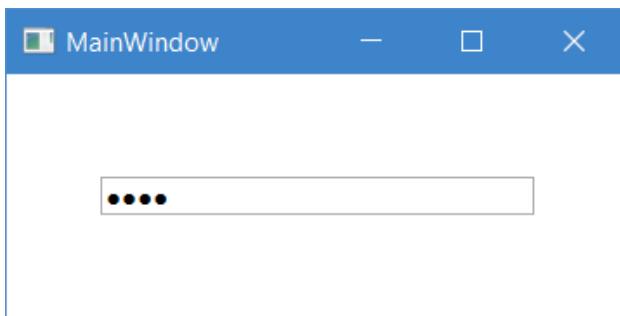


Рис. 42. Элемент управления PasswordBox

Рассмотренной выше разметке соответствует следующий код (полный пример находится в Wpf.Controls.PasswordBox.CSharp):

C#

```
var passwordBox = new PasswordBox
{
    HorizontalAlignment = HorizontalAlignment.Center,
    Password = "Text",
    VerticalAlignment = VerticalAlignment.Center,
    Width = 200.0
};
```

15. Домашнее задание

Задание 1

Необходимо разработать приложение содержащее набор кнопок, занимающих 2/3 ширины окна при любых его размерах (рис. 43). Каждая кнопка должна в качестве содержимого отображать название цвета и обладать наружным отступом равным 2.0. Также соответствующий цвет должен использоваться в качестве цвета переднего плана кнопки. Необходимо использовать следующий набор цветов: Navy, Blue, Aqua, Teal, Olive, Green, Lime, Yellow, Orange, Red, Maroon, Fuchsia, Purple, Black, Silver, Gray, White.



Рис. 43. Задание 1

Задание 2

Необходимо разработать приложение «Калькулятор» (рис. 44). В верхней части приложения необходимо использовать два поля для ввода текста. Первое используется для отображения предыдущих операций, а второе — для ввода текущего числа. Оба поля должны запрещать редактировать свое содержимое посредством клавиатурного ввода.

Данные поля будут заполняться автоматически при нажатии на соответствующие кнопки, расположенные ниже.

Кнопки «0» — «9» добавляют соответствующую цифру в конец текущего числа. При этом должны выполняться проверки, не допускающие неправильного ввода. Например, нельзя вводить числа, начинающиеся с ноля, после которого нет десятичной точки.

Кнопка «.» добавляет (если это возможно) десятичную точку в текущее число.

Кнопки «/», «*», «+», «-» выполняют соответствующую операцию над результатом предыдущей операции и текущим числом.

Кнопка «=» вычисляет выражение и выводит результат.

Кнопка «CE» очищает текущее число.

Кнопка «C» очищает текущее число и предыдущее выражение.

Кнопка «<» очищает последний введенный символ в текущем числе.

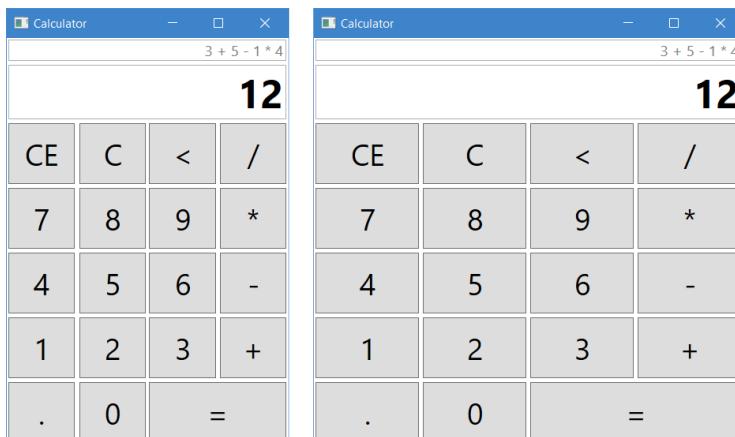


Рис. 44. Задание 2



Урок № 1

Введение в WPF. Контейнеры. Введение в элементы управления

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

Все права на охраняемые авторским правом фото-, аудио- и видеопрограммные средства, 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.