МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Ан. Б. Шамшев

ОСНОВЫ ПРОЕКТИРОВАНИЯ ИНТЕРФЕЙСОВ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ WINDOWS PRESENTATION FOUNDATION

Учебное пособие

Ульяновск
УлГТУ
2012

ББК 32.973.2-018.2 (075)
УДК 004.451
Ш 19

Рецензенты:

кафедра «Информационные технологии» Ульяновского государственного университета (зав. кафедрой канд. физ.-мат. наук, доцент М. А. Волков);

профессор кафедры «Информационные технологии» УлГУ, д-р техн. наук, И. В. Семушин.

Утверждено редакционно-издательским советом университета в качестве учебного пособия

Шамшев, Ан. Б.

Ш 19 Основы проектирования интерфейсов с использованием технологии Windows Presentation Foundation : учебное пособие / Ан. Б. Шамшев. – Ульяновск : УлГТУ, 2012. – 163 с.

ISBN 978-5-9795-0924-2

Представлены базовые технологии проектирования интерфейсов с использованием технологии Windows Presentation Foundation для платформы MicrosoftdotNET. Особенности проектирования интерфейсов иллюстрируются примерами кодов разметки.

Пособие предназначено для студентов направления 230700.62 «Прикладная информатика» профиль «Прикладная информатика в экономике», изучающих дисциплину «Проектирование интерфейсов», студентов направления 231000.62 «Программная инженерия», изучающих дисциплину «Проектирование человеко-машинного интерфейса», а также для студентов других направлений, изучающих дисциплины, связанные с проектированием интерфейсов.

ББК32.973.2-018.2 (075)
УДК 004.451

ISBN 978-5-9795-0924-2

© Шамшев Ан. Б., 2012
© Оформление. УлГТУ, 2012

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ.......... 5
1. ВВЕДЕНИЕ В WINDOWS PRESENTATION FOUNDATION.......... 7
1.1. WPF и DirectX.......... 10
1.2. Новые возможности WPF.......... 12
1.3. Независимость от разрешения монитора.......... 14
2. ОСНОВЫ XAML.......... 19
2.1. Подмножества и основные правила XAML.......... 21
2.2. Пространства имен XAML.......... 24
2.3. Класс отделенного кода и имена элементов.......... 26
2.4. Свойства и события в XAML.......... 28
2.5. Типы из других пространств имен.......... 39
3. КОМПОНОВКА ЭЛЕМЕНТОВ ИНТЕРФЕЙСА.......... 41
3.1. Понятие компоновки в WPF.......... 41
3.2. Процесс компоновки.......... 44
3.3. Компоновка с помощью StackPanel.......... 46
3.4. WrapPanel и DockPanel.......... 54
3.5. Grid.......... 59
3.6. UniformGrid.......... 75
3.7. Координатная компоновка с помощью Canvas.......... 76
3.8. InkCanvas.......... 79
3.9. Примеры компоновки.......... 82
4. СОДЕРЖИМОЕ.......... 89
4.1. Элементы управления содержимым.......... 89
4.2. Свойство Content.......... 92
4.3. Выравнивание содержимого.......... 95
4.4. Модель содержимого в WPF.......... 96
4.5. Специализированные контейнеры.......... 98
4.6. Элементы управления содержимым с заголовком.......... 104
4.7. Элемент управления Expander.......... 107
4.8. Декораторы.......... 112

5. СВОЙСТВА ЗАВИСИМОСТЕЙ И МАРШРУТИЗИРУЕМЫЕ СОБЫТИЯ.. 116
5.1. Маршрутизированные события 117
5.2. Маршрутизация событий 119
5.3. Поднимающиеся события 123
5.4. Прикрепляемые события 126
5.5. События WPF 130
ЗАКЛЮЧЕНИЕ 152
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ.. 153
ГЛОССАРИЙ 154
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 156
ПРИЛОЖЕНИЕ. ПРИМЕРЫ ТЕСТОВЫХ ЗАДАНИЙ 157

ПРЕДИСЛОВИЕ

В учебном пособии представлены базовые понятия технологии проектирования интерфейсов с использованием технологии Windows Presentation Foundation для платформы Microsoft dotNET. Особенности проектирования интерфейсов иллюстрируются примерами кодов разметки.

Пособие предназначено для студентов направления 230700.62 «Прикладная информатика» профиль «Прикладная информатика в экономике», изучающих дисциплину «Проектирование интерфейсов», студентов направления 231000.62 «Программная инженерия», изучающих дисциплину «Проектирование человеко-машинного интерфейса», а также студентов других направлений, интересующихся современными технологиями создания пользовательских интерфейсов.

Выписка из ГОС ВПО:

Цикл, к которому относится дисциплина	Компетенции студента, формируемые в результате освоения дисциплины
Б3.В.1.3	ПК-11: способность принимать участие в создании и управлении ИС на всех этапах жизненного цикла; В результате изучения дисциплины студент должен: • *знать* основы построения пользовательских интерфейсов, основные способы и возможности среды разработки пользовательских интерфейсов, средства создания динамических интерфейсов и привязки интерфейсов к внешним данным; • *уметь* формировать пользовательские интерфейсы при помощи средств разработки и языка описания интерфейсов XAML; • *владеть* средствами разработки пользовательских интерфейсов Visual Studio 2010 и Expression Studio, технологией разработки пользовательских интерфейсов WindowsPresentationFoundation.

В учебном пособии рассматриваются следующие разделы учебной программы: введение в Windows Presentation Foundation, основы XAML, основы компоновки пользовательского интерфейса, обработка событий пользовательского интерфейса.

1. ВВЕДЕНИЕ В WINDOWS PRESENTATION FOUNDATION

Сразу после своего появления платформа Microsoft .NET породила большое количество новых технологий. Это был новый способ написания Web-приложений (ASP.NET), совершенно новый способ подключения к базам данных (ADO.NET), новые языки программирования с безопасностью в отношении типов (в частности, C# и VB.NET) и управляемая исполняющая среда (CLR). Не менее важной среди этих новшеств была Windows Forms – библиотека классов, необходимых для создания Windows-приложений.

Несмотря на то, что Windows Forms является зрелым и полнофункциональным инструментальным средством, она жестко связана с основными конструктивными особенностями Windows, которые не менялись на протяжении последних десяти лет. Более того, Windows Forms основывается на интерфейсе Windows API при создании внешнего вида стандартных элементов пользовательского интерфейса, таких как кнопки, текстовые окна, флажки и т. п. Как результат, эти ингредиенты фактически не поддаются настройке.

Все поменялось благодаря новой модели создания пользовательских интерфейсов, которую предлагает новая технология создания пользовательских интерфейсов Windows Presentation Foundation (WPF). Несмотря на то, что WPF включает стандартные элементы управления, она сама рисует каждый текст, рамку и фон. Как результат, WPF может предложить гораздо больше мощных функций, которые могут изменить любой элемент содержимого, визуализируемого на экране. С помощью этих функций можно изменить стиль обычных элементов управления, таких как кнопки, зачастую без необходимости переписывания кода. Также можно использовать объекты трансформации, чтобы вращать, растягивать, изменять масштаб и искажать все, что относится к пользовательскому интерфейсу. Можно использовать встроенную систему анимации в WPF, чтобы все это дела-

лось в динамике. Поскольку механизм WPF визуализирует содержимое окна как часть одной операции, он может обрабатывать неограниченное число слоев перекрытия элементов управления, даже если они имеют нестандартные формы или частичную прозрачность.

В основе новых возможностей WPF лежит мощная новая инфраструктура, основанная на DirectX-API – интерфейсе аппаратного ускорения графики, который обычно используется в современных компьютерных играх. Это означает, что можно применять богатые графические эффекты без ущерба для производительности, как это было бы при использовании Windows Forms. В действительности, можно даже получить расширенные функции, такие как поддержка видеофайлов и трехмерного содержимого. С их помощью можно создавать великолепные пользовательские интерфейсы и визуальные эффекты, чего невозможно добиться с помощью Windows Forms.

Несмотря на то, что современные функции видео, анимации и трехмерных изображений часто становятся объектом наибольшего внимания в WPF, важно отметить, что можно применять WPF и для создания традиционных Windows приложений со стандартными элементами управления и простым внешним видом. В действительности, совсем несложно использовать обычные элементы управления в WPF – точно так же, как и в Windows Forms. Более того, WPF улучшает функции, которые будут представлять интерес для разработчиков бизнес-приложений, включая существенно улучшенную модель привязки данных, новый набор классов для печати содержимого и управления очередью печати, а также возможность использования потоковых документов для отображения больших объемов форматированного текста. WPF предоставляет новую модель для создания страничных приложений, выполняющихся в браузере и которые могут запускаться с Web-сайта.

Наконец, WPF комбинирует лучшие качества из старого мира

разработки приложений для Windows и новые инновационные технологии для создания современных, насыщенных качественной графикой пользовательских интерфейсов.

Данное учебное пособие рассчитано на использование последней версии платформы – .NET 4 (на момент написания учебного пособия). Однако большинство описанных концепций справедливо и для более ранних версий .NET - 3.0 и 3.5.

Для того чтобы запускать приложения WPF, компьютер должен работать под управлением одной из следующих операционных систем: Microsoft Windows 7, Microsoft Windows Vista, или Microsoft Windows XP с пакетом обновлений Service Pack 3. Также должен быть установленасреда выполнения .NET Framework 4.Чтобы создавать приложения WPF 4, необходима среда Visual Studio 2010, включающая .NET Framework 4. Однако возможен и другой вариант: вместо того, чтобы устанавливать Visual Studio, можно использовать Expression Blend – графически ориентированный инструмент проектирования, предназначенный для создания и тестирования WPF-приложений. В данном учебном пособии предполагается использование Visual Studio.

WPF – совершенно новая графическая система отображения для Windows.WPF спроектирована для .NET под влиянием таких современных технологий отображения, как HTML и Flash, с использования аппаратного ускорения. Она также представляет собой наиболее радикальное изменение в пользовательском интерфейсе Windows со времен Windows 95.

Сложно переоценить важность WPF, не зная, что разработчики Windows использовали одну и туже технологию отображения в течение более 15 лет. Стандартное приложение Windows для создания пользовательского интерфейса использует две основополагающие части операционной системы Windows:

- User32 обеспечивает знакомый внешний вид и поведение таких элементов, как окна, кнопки, текстовые поля и т. п.;
- GDI/GDI+ предоставляет поддержку рисования фигур, текста.

С годами обе технологии совершенствовались, и API-интерфейсы, используемые разработчиками для взаимодействия с ними, значительно изменились. Но вне зависимости от технологии разработкивсегда использовались одни и те же части операционной системы Windows. Новые каркасы просто предоставляли лучшие оболочки для взаимодействия с User32 и GDI/GDI+. Они могут быть более эффективными, менее сложными, однако они не могли преодолеть фундаментальные ограничения системных компонентов, разработанных более 10 лет назад.

1.1. WPF и DirectX

В Microsoft разработали способ преодоления ограничений, присущих библиотекам User32 и GDI/GDI+-DirectX. DirectX создавался как инструментарий для создания игр на платформе Windows. Главной его целью была скорость, и потому Microsoft тесно сотрудничала с производителями видеокарт, чтобы обеспечить для DirectX аппаратную поддержку, необходимую для отображения сложных текстур, специальных эффектов, таких как частичная прозрачность и т. д. На сегодняшний день DirectX является неотъемлемой частью Windows, которая включает поддержку всех современных видеокарт. Однако программный интерфейс DirectX по-прежнему несет в себе наследие своих корней как средства разработки игр. Из-за присущей DirectX сложности он почти никогда не использовался в традиционных приложениях Windows.

WPF кардинально изменяет сложившуюся ситуацию. Лежащая в основе WPF графическая технология – это не GDI/GDI+, а DirectX. Приложения WPF используют DirectX независимо от создаваемого типа пользовательского интерфейса, т. е. вся работа по рисованию

интерфейса проходит через конвейер DirectX. В результате даже самые заурядные бизнес-приложения могут использовать богатые эффекты вроде прозрачности и сглаживания. Также используется аппаратноеускорение, и это означает, что DirectX передает как можнобольше работы GPU. Эффективность DirectX объясняется тем, что он оперирует высокоуровневыми ингредиентами вроде текстур и градиентов, которые могут отображаться непосредственно видеокартой. GDI/GDI+ на это не способен, поэтому ему приходится конвертировать их в инструкции рисования пикселей, и потому на современных видеокартах отображение идет намного медленнее.

Один компонент GDI/GDI+, который по-прежнему используется – это User32. Это объясняется тем, что WPF по-прежнему полагается на User32 в отношении таких служб, как обработка и маршрутизация ввода, а также определение того, какое приложение владеет какой частью экрана. Однако все рисование осуществляется через DirectX.

Видеокарты различаются между собой в их поддержке специализированных средств визуализации и оптимизации. При программировании с DirectX это является существенной проблемой. С применением WPF она не так сильно проявляется, поскольку WPF обладает способностью выполнять всю работу с использованием программных вычислений вместо того, чтобы полагаться на встроенную поддержку видеокарты. Существует одно исключение в отношении программной поддержки WPF. Из-за слабой поддержки драйверов WPF выполняет сглаживание трехмерной графики только в случае, если приложение запущено под WindowsVista, или Windows 7. Но сглаживание всегда обеспечивается для двумерной графики, независимо от операционной системы и поддержки драйверов.

Если бы единственным достоинством WPF было аппаратное ускорение через DirectX, это уже было бы значительным усовершенствованием. Однако WPF на самом деле включает целый набор высокоуровневых служб, ориентированных на прикладных программистов.

1.2. Новые возможности WPF

Ниже приведен список некоторых наиболее существенных изменений, которые принес с собой WPF в мир программирования Windows:

- Web-подобная модель компоновки. Вместо того чтобы фиксировать элементы управления на месте с определенными координатами, WPF поддерживает гибкий поток, размещающий элементы управления на основе их содержимого. В результате получается пользовательский интерфейс, который может быть адаптирован для отображения высокодинамичного содержимого или разных языков;
- Богатая модель рисования. Вместо рисования пикселей в WPF имеет дело с примитивами – базовыми фигурами, блоками текста и другими графическими элементами. Также имеются такие новые средства, как действительно прозрачные элементы управления, возможность складывать множество уровней с разной степенью прозрачности, а также встроенную поддержку трехмерной графики;
- Богатая текстовая модель. После многих лет нестандартной обработки текстов в таких несовершенных элементах управления, как классический Label,WPF предоставляет приложениям Windows возможность отображения богатого стилизованного текста в любом месте пользовательского интерфейса. И если необходимо отображать значительные объемы текста, можно воспользоваться развитыми средствами отображения документов, такими как переносы, разбиение на колонки и выравнивание для повышения читабельности;
- Анимация как программная концепция. В WPF анимация – неотъемлемая часть программного каркаса. Она определяется декларативными дескрипторами, и WPF запускает ее в действие автоматически;

- Поддержка аудио и видео. Прежние инструментарии пользовательского интерфейса, такие как WindowsForms, были весьма ограничены в работе с мультимедиа. Но WPF включает поддержку воспроизведения любого аудио- или видеофайла, поддерживаемого WindowsMediaPlayer, позволяя воспроизводить более одного медиафайла одновременно. Такжепредоставляются инструменты для интеграции аудио- и видеоданных в остальную часть пользовательского интерфейса;
- Стили и шаблоны. Стили позволяют стандартизовать форматирование и повторно использовать его по всему приложению. Шаблоны позволяют изменить способ отображения элементов, даже таких основополагающих, как кнопки;
- Команды. Большинство пользователей знают, что не имеет значения, откуда они инициируют команду Open (Открыть) – через меню или панель инструментов; конечный результат одинаков. Теперь эта абстракция доступна на уровне кода – можно определять прикладные команды в одном месте и привязывать их к множеству элементов управления;
- Декларативный пользовательский интерфейс. Хотя можно конструировать окно WPF в программном коде, VisualStudio использует другой подход. Содержимое каждого окна описывается в виде XML-дескрипторов в документе XAML. Преимущество такого подхода состоит в том, что пользовательский интерфейс полностью отделен от кода, и графические дизайнеры могут использовать профессиональные инструменты для редактирования файлов XAML, улучшая внешний вид всего приложения;
- Приложения на основе страниц. Используя WPF, можно строить браузер – подобные приложения, которые позволяют перемещаться по коллекции страниц, оснащенной кнопками навигации вперед и назад. WPF автоматически обрабатывает все сложные детали, такие как хронология посещения страниц.

Существует возможность даже развернуть проект в виде браузерного приложения, которое выполняется внутри браузера с установленным дополнением.

1.3. Независимость от разрешения монитора

Традиционные приложения Windows связаны определенными предположениями относительно разрешения экрана. Обычно разработчики рассчитывают на стандартное разрешение монитора (например, 1024×768 пикселей) и проектируют свои окна с учетом этого, стараясь обеспечить разумное поведение при изменении размеров в большую и меньшую сторону.

Проблема состоит в том, что пользовательский интерфейс в традиционных приложениях Windows не является масштабируемым. В результате, при использовании монитора высокого разрешения, который располагает пиксели более плотно, окно приложения становится меньше и читать его труднее. Эта проблема особенно актуальна для новых мониторов и смартфонов, которые имеют высокую плотность пикселей или работают с более высоким разрешением.

WPF не страдает от этой проблемы, потому что самостоятельно визуализирует все элементы пользовательского интерфейса. Когда происходит работа с обычными элементами управления, то можно рассчитывать на независимость WPF от разрешения. WPF автоматически заботится о том, чтобы все имело правильный размер. Однако если планируется включать в приложение изображения, такой уверенности может уже не быть. Например, в традиционных приложениях Windows разработчики используют крошечные растровые изображения для команд панели инструментов. В приложении WPF такой подход не идеален, потому что битовая карта может отображать размытые артефакты, которые будут масштабироваться вверх и вниз согласно системной установке DPI. Вместо этого при проектировании пользовательского интерфейса WPF даже самые мелкие пиктограммы обычно реализованы в векторной графике.

WPF – платформа будущего для разработки пользовательского

интерфейса Windows. Однако она не заменит полностью Windows-Forms. Во многих отношениях WindowsForms – это кульминация технологии отображения, построенной на GDI/GDI+ и User32. Это более зрелая технология, чем WPF, и она все еще включает средства, которые пока еще не нашли своего места в инструментарии WPF.

Возникает следующий вопрос – какую платформу следует выбрать для создания нового приложения Windows? Если проект начинается с нуля или нужно одно из средств, представленных в WPF, но отсутствующих в WindowsForms, WPF – идеальный выбор, которому обеспечены наилучшие перспективы в отношении расширяемости и долгожительства. С другой стороны, если сделаны существенные вложения в бизнес-приложение на основе WindowsForms, то нет необходимости переносить его на WPF. Платформа WindowsForms будет поддерживаться еще долгие годы.

Есть одна область, где WPF пока далек от идеала: когда нужно создавать приложения со строгими требованиями к графике реального времени, вроде сложных симуляторов физических процессов или современных игр. Если необходимо получить максимально возможную производительность видео для приложений подобного рода, то придется программировать на значительно более низком уровне, используя так называемый «сырой» DirectX. Управляемые библиотеки .NET для программирования с DirectX можно загрузить с сайта http://msdn.microsoft.com/ru-ru/directx.

Подобно самому .NETFramework, WPF – это технология, основанная на Windows. Это означает, что приложения WPF могут использоваться только на компьютерах под управлением операционной системы Windows. Браузер-ориентированные приложения WPF столь же ограничены – они могут работать только на компьютерах под управлением Windows.

WPF использует многослойную архитектуру, представленную на рис. 1:

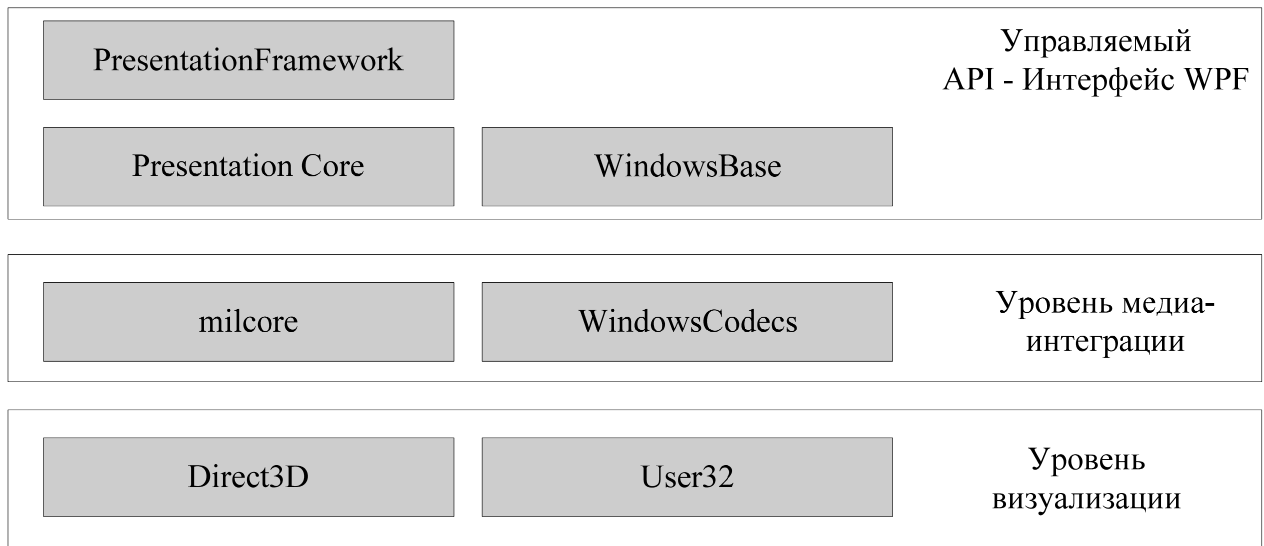

Рис. 1. Архитектура WPF

На рис. 2 показаны некоторые ключевые ветви иерархии классов:

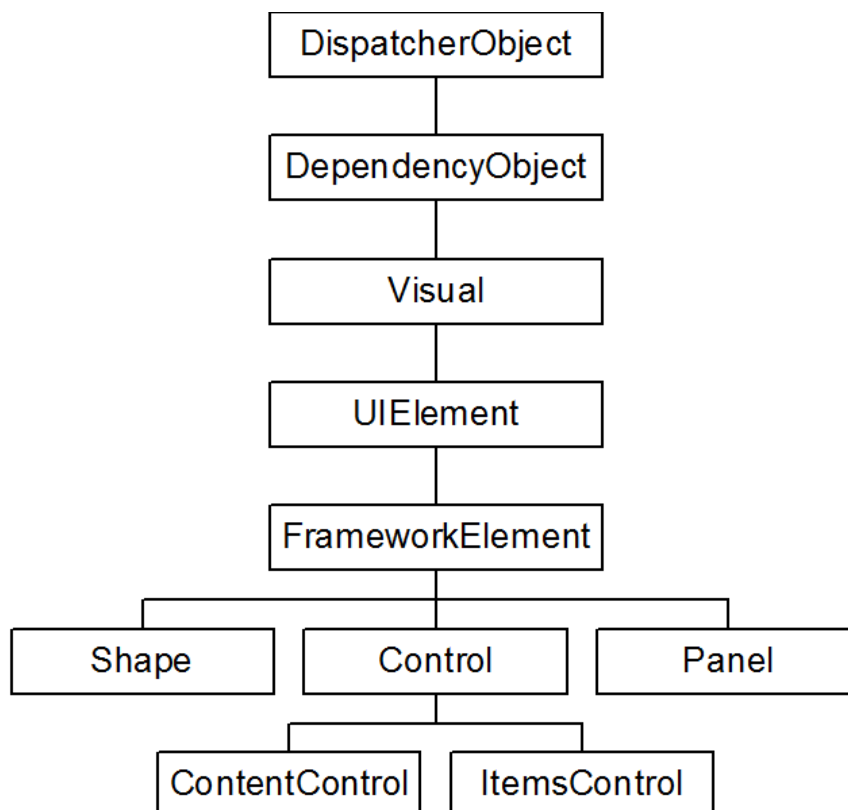

Рис. 2. Базовые классы WPF

В последующих разделах будут описаны основные классы из этой диаграммы. Многие из них ведут к целым ветвям элементов (таких как фигуры, панели и элементы управления). Кратко опишем основные классы:

- ***System.Threading.DispatcherObject:*** Приложения WPF используют однопоточную модель, а это означает, что весь пользовательский интерфейс принадлежит единственному потоку. Взаимодействовать с элементами пользовательского интерфейса из других потоков небезопасно. Чтобы содействовать работе этой модели, каждое приложение WPF управляется диспетчером, координирующим сообщения. Будучи унаследованным от DispatcherObject, каждый элемент пользовательского интерфейса может удостовериться, выполняется ли код в правильном потоке, и обратиться к диспетчеру, чтобы направить код в поток пользовательского интерфейса;
- ***System.Windows.DependencyObject:*** В WPF центральный путь взаимодействия с экранными элементами проходит через свойства. На ранней стадии цикла проектирования архитекторы WPF решили создать более мощную модель свойств, которая положена в основу таких средств, как уведомления об изменениях, наследуемые значения по умолчанию и более экономичное хранилище свойств. Конечным результатом стало средство свойств зависимости (dependency property). Наследуясь от DependencyObject, классы WPF получают поддержку свойств зависимости;
- ***System.Windows.Media.Visual:*** Каждый элемент, появляющийся в WPF, в основе своей является Visual. Можно воспринимать класс Visual как единственный объект рисования, инкапсулирующий в себе базовые инструкции рисования, дополнительные возможности рисования и базовую функциональность. Любой класс, унаследованный от Visual, обладает способностью отображаться в окне;
- ***System.Windows.UIElement:*** UIElement добавляет поддержку таких сущностей WPF, как компоновка (layout), ввод (input),

фокус (focus) и события (events) – все, что команда разработчиков WPF называет аббревиатурой LIFE. Как и со свойствами, WPF реализует расширенную систему передачи событии, именуемуюмаршрутизируемыми событиями;

- ***System.Windows.FrameworkElement:*** FrameworkElement – конечный пункт в центральном дереве наследования WPF. Он реализует некоторые члены, которые просто определены в UIElement;
- ***System.Windows.Shapes.Shape:*** От этого класса наследуются базовые фигуры, такие как Rectangle, Polygon, Ellipse, Line и Path. Эти фигуры могут быть использованы наряду с более традиционными визуальными элементами Windows вроде кнопок и текстовых полей;
- ***System.Windows.Controls.Control:*** Элемент управления (control) – это элемент, который может взаимодействовать с пользователем. К нему, очевидно, относятся такие классы, как TextBox,Button и ListBox. Класс Control добавляет дополнительные свойства для установки шрифта, а также цветов переднего плана и фона. Но наиболее интересная деталь, которую он предоставляет – это поддержка шаблонов, которая позволяет заменять стандартный внешний вид элемента управления собственным рисованием;
- ***System.Windows.Controls.ContentControl:*** Это базовый класс для всех элементов управления, которые имеют отдельный блок содержимого. Сюда относится все – от скромной метки Label до окна Window. Наиболее впечатляющая часть этой модели заключается в том, что единственный кусок содержимого может быть чем угодно – от обычной строки до панели компоновки, содержащей комбинацию других фигур и элементов управления;

- ***System.Windows.Controls.ItemsControl:*** Это базовый класс для всех элементов управления, которые отображают коллекцию каких-то единиц информации, вроде ListBox и TreeView. Фактически, в WPF все меню, панели инструментов и линейки состояния на самом деле являются специализированными списками, и классы, реализующие их, наследуются от ItemsControl;
- ***System.Windows.Controls.Panel:*** Это базовый класс для всех контейнеров компоновки – элементов, которые содержат в себе один или более дочерних элементов и упорядочивают их в соответствии с определенными правилами компоновки. Эти контейнеры образуют фундамент системы компоновки WPF, и их использование – ключ к упорядочиванию содержимого наиболее привлекательным и гибким способом.

2. ОСНОВЫ XAML

XAML представляет собой язык разметки, используемый для создания экземпляров объектов .NET. Хотя язык XAML – это технология, которая может быть применима ко многим различным предметным областям, его основное назначение – конструирование пользовательских интерфейсов WPF. Другими словами, документы XAML определяют расположение панелей, кнопок и прочих элементов управления, составляющих окна в приложении WPF.

Код XAMLредко приходится писать вручную. Вместо этого обычно используется инструмент, генерирующий необходимый код XAML. Для графического дизайнера таким инструментом, скорее всего, будет программа рисования и графического, например Microsoft Expression Blend. Для разработчика это наверняка Visual Studio. Поскольку оба инструмента поддерживают XAML, можно создать базовый пользовательский интерфейс в Visual Studio, а затем

передать его команде дизайнеров, которые доведут его до совершенства, добавив специальную графику в Expression Blend. Фактически такая способность интегрировать рабочий поток разработчиков и дизайнеров – одна из ключевых причин создания языка XAML.

Разумеется, при проектировании приложениянет необходимости писать XAML вручную. Вместо этого следует использовать инструмент, подобный VisualStudio, чтобы создать нужное окно методом перетаскивания. Несмотря на то, что код XAML создается средойразработки автоматически, понимание XAML чрезвычайно важно при разработке дизайна приложения WPF. В этом отношении приложения WPF существенно отличаются от приложений Windows Forms. Понимание XAML поможет разобраться с ключевыми концепциями WPF, такими как прикрепленные свойства, компоновка, модель содержимого, маршрутизируемые события и т. д. Что более важно – есть целый ряд задач, решение которых возможно только с помощью вручную написанного XAML, либо существенно облегчается. К ним относятся перечисленные ниже:

- Привязка обработчиков событий. Прикрепление обработчиков событий в наиболее распространенных местах – например, к событию Click для Button – легко сделать в Visual Studio. Однако XAML позволяет создавать более изощренные соединения. Например, можно установить обработчик события, реагирующий на событие Click в каждой кнопке окна;
- Определение ресурсов. Ресурсы – это объекты, которые определены однажды в коде XAML, в специальном разделе, а затем повторно используются в разных местах кода разметки. Ресурсы позволяют централизовать и стандартизовать форматирование и создание невизуальных объектов, таких как шаблоны и анимации;
- Определение шаблонов элементов управления. Элементы

управления WPF проектируются как лишенные внешнего вида; это значит, что можно подставлять собственные визуальные представления элементов вместо стандартных. Чтобы сделать это, необходимо создать собственный шаблон элемента управления, который представляет собой не что иное, как блок разметки XAML;

- Написание выражений привязки данных. Привязка данных позволяет извлекать данные из объекта и отображать их в привязанном элементе. Чтобы установить это отношение и конфигурировать его работу, необходимо добавить выражение привязки данных к коду разметки XAML;
- Определение анимаций. Анимации – распространенный компонент приложений XAML. Обычно они определяются как ресурсы, конструируются с использованием разметки XAML, а затем привязываются к другим элементам управления (или инициируются в коде).

Большинство разработчиков WPF используют комбинацию приемов, разрабатывая часть пользовательского интерфейса с помощью инструмента проектирования (Visual Studio или Expression Blend), а затем проводя тонкую настройку посредством редактирования кода разметки вручную.

2.1. Подмножества и основные правила XAML

Существует несколько разных способов использования термина XAML. Выше он применялся для того, чтобы ссылаться на весь язык XAML, предлагающий основанный на XML синтаксис для представления дерева объектов .NET.

Существует несколько подмножеств XAML, перечисленных ниже:

- WPFXAML включает элементы, описывающие содержимое WPF вроде векторной графики, элементов управления и документов. В настоящее время это наиболее важное применение XAML;

- XPSXAML – часть WPF XAML, определяющая XML-представление форматированных электронных документов. Она опубликована как отдельный стандарт XML Paper Specification (XPS);
- Silverlight XAML – подмножество WPF XAML, предназначенное для Silverlight-приложений. Silverlight – это межплатформенный браузерный подключаемый модуль, позволяющий создавать богатое Web-содержимое с двумерной графикой, анимацией, аудио и видео;
- WWF XAML включает элементы, описывающие содержимое Windows Workflow Foundation (WF).

Стандарт XAML достаточно очевиден, если понять несколько его основополагающих правил:

- Каждый элемент в документе XAML отображается на экземпляр класса .NET. Имя элемента соответствует имени классав точности. Например, элемент <Button> сообщает WPF, что должен быть создан объект Button;
- Как и любой документ XML, код XAML допускает вложение одного элемента внутрь другого. XAML дает каждому классу гибкость в принятии решения относительно того, как справиться с конкретной ситуацией. Однако вложение обычно является способом выразить включение (containment);
- Можно устанавливать свойства каждого класса через атрибуты. Однако в некоторых ситуациях атрибуты не достаточно мощны, чтобы справиться с этой работой. В этих случаях используются вложенные дескрипторы со специальным синтаксисом.

Создадим простейший документ XAML., представляющий новое пустое окно (как оно создано в Visual Studio). Созданный код XAML приведен ниже:

```
<Windowx:Class="WpfApplication1.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525">
<Grid>
</Grid>
</Window>
```

Этот документ включает всего два элемента – элемент верхнего уровня Window, который представляет все окно, и Grid, куда можно поместить элементы управления. Хотя можно использовать любой элемент верхнего уровня, приложение WPF полагается только на несколько из них:

- Window;
- Page (похож на Window, используется для приложений с навигацией);
- Application (определяет ресурсы приложения и начальные установки).

Как во всех документах XML, может существовать только один элемент верхнего уровня. В предыдущем примере это означает, что как только закрывается элемент Window дескриптором </window>, документ завершается. Никакое дополнительное содержание уже не допускается.

В стартовом дескрипторе элемента Window можно найти несколько интересных атрибутов, включая имя класса и два пространства имен XML (описанных в последующих разделах), а также три свойства, показанных ниже:

```
Title="MainWindow" Height="350" Width="525">
```

Каждыйатрибут соответствует отдельному свойству в классе Window. В данном случае приведенный фрагмент кода сообщает WPF о необходимости создать окно с заголовком MainWindow размером 525×350 единиц (размеры зависят от разрешения монитора).

2.2. Пространства имен XAML

Очевидно, что недостаточно просто указать имя класса. Анализатору XAML также нужно знать пространство имен .NET, где находится этот класс. Например, класс Window может находиться в нескольких пространствах имен – он может ссылаться на класс System.Windows.Window, на класс в компоненте от независимого разработчика, или же на класс, определенный в разрабатываемом приложении. Чтобы определить, какой именно класс нужен на самом деле, анализатор XAML проверяет пространство имен XML, к которому относится элемент.

В примере документа, показанном ранее, определено два пространства имен:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Пространства имен объявляются посредством атрибутов. Эти атрибуты могут помещаться внутри начального дескриптора любого элемента. Однако согласно принятым соглашениям, все пространства имен, которые нужно использовать в документе, должны быть объявлены в самом первом дескрипторе

Атрибут xmlns – это специализированный атрибут в XML, который зарезервирован для объявления пространств имен. Этот фрагмент кода разметки объявляет два пространства имен, которые будут присутствовать в каждом создаваемом документе WPFXAML:

- http://schemas.microsoft.com/winfx/2006/xaml/presentation– основное пространство имен WFP. Оно охватывает все классы WPF, включая элементы управления, которые применяются для построения пользовательских интерфейсов;
- http://schemas.microsoft.com/winfx/2006/xaml – пространство имен XAML. Оно включает различные служебные свойства XAML, которые позволяют влиять на то, как ин-

терпретируется документ. Это пространство имен отображается на префикс x, что позволит применять его элементы, помещая префикс пространства имен перед именем элемента (как в <x:ИмяЭлемента>).

Таким образом, пространство имен XML не соответствует никакому конкретному пространству имен .NET. Есть несколько причин, по которым создатели XML выбрали такой дизайн. По существующему соглашению пространства имен XML часто имеют форму URI. Эти URI выглядят так, будто указывают на некоторое место в Web, хотя на самом деле это не так. Формат URI используется потому, что он делает маловероятным ситуацию, когда разные организации нечаянно создадут разные языки на базе XML с одинаковым пространством имен. Поскольку домен schemas.microsoft.com принадлежит Microsoft, только Microsoft использует его в названии пространства имен XML.

Другая причина того, что нет однозначного отображения между пространствами имен XML, используемым в XAML, и пространствами имен .NET заключается в том, что это могло бы значительно усложнить документы XAML. Проблема состоит в том, что WPF включает в себя свыше десятка пространств имен. Если бы каждое пространство имен .NET отображалось на отдельное пространство имен XML,то пришлось бы специфицировать правильное пространство имен для любого используемого элемента управления, что быстро привело бы к путанице. Вместо этого создатели WPF предпочли комбинировать все эти пространства имен .NET в единое пространство имен XML.

Информация пространства имен позволяет анализатору XAML находить правильный класс. Например, когда он обрабатывает элементы Window и Grid, он определяет, что они помещены в пространство имен WPF по умолчанию. Затем он ищет соответствующие про-

странства имен .NET – и находит System.Windows.Window и System.Windows.Controls.Grid.

2.3. Класс отделенного кода и имена элементов

XAML позволяет конструировать пользовательский интерфейс, но для того, чтобы создать функционирующее приложение, нужен способ подключения обработчиков событий, содержащих код приложения.XAML позволяет легко это сделать с помощью атрибута Class, показанного ниже:

```
<Window x:Class="WpfApplication1.MainWindow"
```

Префикс пространства имен x помещает атрибут Class в пространство имен XAML, что означает более общую часть языка XAML. Фактически атрибут Class сообщает анализатору XAML, чтобы он сгенерировал новый класс с указанным именем. Этот класс наследуется от класса, именованного элементом XML. Другими словами, этот пример создает новый класс по имени MainWindow, который наследуется от базового класса Window.

Класс MainWindow генерируется автоматически во время компиляции. Можно предоставить часть класса MainWindow, которая будет объединена с автоматически сгенерированной частью этого класса. Специфицированная разработчиком часть – хороший контейнер для кода обработки событий. Подобное разделение возможно благодаря средству языков .NET, известному под названиемчастичных классов. Частичные классы позволяют разделить класс на две или более отдельных части во время разработки, которые соединяются вместе в компилированной сборке. Частичные классы могут быть использованы во многих сценариях управления кодом, но более всего удобны, когда код должен объединяться с файлом, сгенерированным дизайнером.

Среда VisualStudio помогает разработчику за счет автоматического создания частичного класса, куда он может поместить свой код

обработки событий. Например, в показанном выше примере среда разработки автоматически сгенерировала следующий программный код:

```
namespace WpfApplication1
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
public partial class MainWindow : Window
{
publicMainWindow()
        {
InitializeComponent();
        }
    }
}
```

При компиляции приложения XAML, определяющий пользовательский интерфейс, транслируется в объявление типа CLR, объединенного с логикой файла класса отделенного кода, формируя один общий модуль.

Отметим важную деталь, касающуюся имен объектов. В классе отделенного кода часто возникает желание программно манипулировать элементами управления. Например, можно читать либо изменять свойства, прикреплять или откреплять обработчики событий в процессе выполнения. Чтобы обеспечить такую возможность, элемент управления должен включать XAML-атрибут Name. В предыдущем примере элемент Grid не включает атрибут Name, поэтому нельзя манипулировать им в отделенном коде. Следующий код присваивает имя элементу Grid:

```
<Gridx:Name="Grid1">

</Grid>
```

Это действие можно провести в документе XAML вручную, или же через установку свойства Name в окне Properties (Свойства) Visual Studio.Теперь можно взаимодействовать с элементом в коде класса MainWindow, используя имя Grid1:

```
MessageBox.Show(String.Format("The grid is {0}x{1} units in
size.", Grid1.ActualWidth, Grid1.ActualHeight));
```

Свойство Name, показанное выше, является частью языка XAML и используется для того, чтобы помочь в интеграции класса отделенного кода.

2.4. Свойства и события в XAML

Рассмотренный выше пример был очень простым – пустое окно, содержащее пустой элемент управления Grid. В следующем примере рассмотрим более реалистичное окно, включающее несколько элементов управления.

На рис. 3 показан пример с автоответчиком на вопросы пользователя:

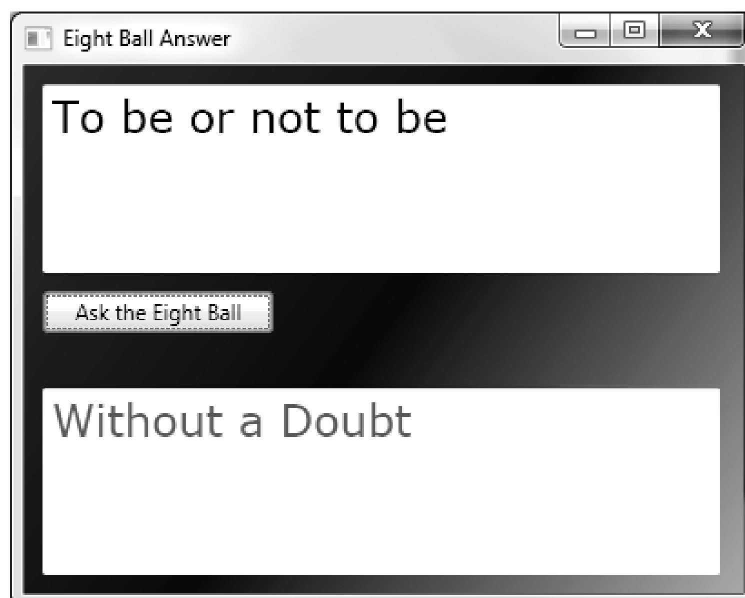

Рис. 3. Окно автоответчика

Окно автоответчика включает четыре элемента управления: Grid, два объекта TextBox и один Button. Разметка, которая необходима для компоновки и конфигурирования этих элементов управле-

ния, существенно длиннее, чем в предыдущих примерах. Ниже приведен сокращенный листинг, в котором некоторые детали заменены многоточиями для демонстрации общей структуры:

```
<Window x:Class="EightBall.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Eight Ball Answer" Height="328" Width="412" >
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="*" />
<RowDefinition Height="Auto" />
<RowDefinition Height="*" />
</Grid.RowDefinitions>
<TextBox …>
      [Place question here.]
</TextBox>
<Button …>
      Ask the Eight Ball
</Button>
<TextBox …>
      [Answer will appear here.]
</TextBox>
</Grid>
</Window>
```

Части этого документа будут рассмотрены ниже.

2.4.1. Простые свойства и конвертеры типов

Как уже было отмечено, атрибуты элемента устанавливают свойства соответствующего объекта. Например, текстовые поля в примере автоответчика конфигурируют выравнивание, поля и шрифт:

```
<TextBox VerticalAlignment="Stretch" HorizontalAlign-
ment="Stretch" Margin="10,10,13,10" Name="txtQuestion"
TextWrapping="Wrap" FontFamily="Verdana" FontSize="24"
```

```
Grid.Row="0" >
      [Place question here.]
</TextBox>
```

Чтобы этот код заработал, класс System.Windows.Controls.TextBox должен содержать следующие свойства: VerticalAlignment, HorizontalAlignment, FontFamily, FontSize и Foreground. Значения этих свойств будут объяснены ниже.

Чтобы заставить эту систему работать, анализатор XAML должен выполнить больше работы, чем может показаться. Значение в атрибуте XAML всегда представлено простой строкой. Однако свойства объекта могут быть любого типа .NET. В примере выше было два свойства, использующих перечисления (VerticalAlignment и HorizontalAlignment), одна строка (FontFamily), одно целое число (FontSize) и один объект Brush (Foreground).

Чтобы преодолеть различие между строковыми значениями и не строковыми свойствами, анализатору XAML необходимо выполнить преобразование. Это преобразование осуществляетсяконвертерами типов – базовой частью инфраструктуры .NET.

Фактически, конвертер типов играет только одну роль – он предоставляет служебные методы, которые могут преобразовывать определенный тип данных .NET в любой другой тип .NET, такой как строку в данном случае. Анализатор XAML выполняет следующие два шага, чтобы найти нужный конвертер типа:

- Проверяет объявление свойства в поисках атрибута TypeConverter. Например, когда приииспользовании свойства Foreground, .NET проверяет объявление свойства Foreground;
- Если в объявлении свойства отсутствует атрибут TypeConverter, то анализатор XAML проверяет объявление класса соответствующего типа данных. Например, свойство Foreground использует объект Brush. Класс Brush использует BrushConverter, потому что класс Brush оснащен объявлением атрибута TypeConverter (type of (BrushConverter)).

Если в объявлении свойства или объявлении класса не оказывается ассоциированного конвертера типа, то анализатор XAML генерирует ошибку.

2.4.2. Сложные свойства

Как бы ни были удобны конвертеры типов, они подходят не для всех случаев. Например, некоторые свойства являются полноценными объектами со своими собственными наборами свойств. Хотя можно создать строковое представление, которое будет использовать конвертер типа, этот синтаксис может быть сложен в применении и подвержен ошибкам.

Для таких случаевXAML предусматривает другой выбор – синтаксис «свойство-элемент». С помощью этого синтаксиса можно добавлять дочерний элемент с именем в форме «Родитель.ИмяСвойства». Например, у Grid есть свойство Background, которое позволяет применять кисть, используемую для рисования области, находящейся под элементами управления. Если требуется использовать сложную кисть – более совершенную, чем сплошное заполнение цветом – то для этого нужно добавить дочерний дескриптор по имени Grid.Background, как показано ниже:

```
<Grid>
<Grid.Background>
</Grid.Background>
</Grid>
```

Идентификации сложного свойства недостаточно для его применения. Для этого необходимо установить его значение. Для этого нужно добавить другой дескриптор, чтобы создать экземпляр определенного класса. В примере с автоответчиком, показанном на рис. 3, фон заливается градиентом. Чтобы определить нужный градиент, следует создать объект LinearGradientBrush.

В соответствии с правилами XAML, для этого следует создать

объект LinearGradientBrush, используя элемент по имени LinearGradientBrush:

```
<Grid>
<Grid.Background>
<LinearGradientBrush>
</LinearGradientBrush>
</Grid.Background>
</Grid>
```

LinearGradientBrush является частью набора пространств имен WPF, поэтомуможно использовать пространство имен XML по умолчанию для дескрипторов.Однако недостаточно просто создать LinearGradientBrush, также необходимо указать цвета градиента. Это делается заполнением свойства LinearGradientBrush.GradientStops коллекцией объектов GradientStop. Свойство GradientStops также является сложным, чтобы быть установленным только одним значением атрибута:

```
<Grid>
<Grid.Background>
<LinearGradientBrush>
<LinearGradientBrush.GradientStops>
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Grid.Background>
</Grid>
```

И, наконец, нужно заполнить коллекцию GradientStops серией объектов GradientStop. Каждый объект GradientStop имеет свойства Offset и Color, применять которые можно, используя обычный синтаксис «свойство-элемент»:

```
<Grid>
<Grid.Background>
<LinearGradientBrush>
<LinearGradientBrush.GradientStops>
<GradientStop Offset="0.00" Color="Red" />
```

```
<GradientStop Offset="0.50" Color="Indigo" />
<GradientStop Offset="1.00" Color="Violet" />
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Grid.Background>
</Grid>
```

2.4.3. Расширения разметки

Для большинства свойств синтаксис свойств XAML работает исключительно хорошо. Но в некоторых случаях просто невозможно жестко закодировать значение свойства. Например, можно установить значение свойства в уже существующий объект. Или же требуется устанавливать значение свойства динамически, привязывая его к свойству в другом элементе управления. В обоих таких случаях нужно использовать расширение разметки (markup extension) – специализированный синтаксис, устанавливающий свойство нестандартным образом.

Расширения разметки могут применяться во вложенных дескрипторах или атрибутах XML, что встречается чаще. Когда они используются в атрибутах, то всегда окружаются фигурными скобками {}. Например, следующим образомможно использовать StaticExtension, позволяющее сослаться на статическое свойство другого класса:

```
<Button Foreground="{x:Static SystemCo-
lors.ActiveCaptionBrush}">
```

Расширения разметки используют синтаксис {КлассРасширенияРазметки Аргумент}. В этом случае расширением разметки служит класс StaticExtension. Префикс x: указывает на то, что StaticExtension находится в одном из пространств имен XAML. Также встречаются расширения разметки, являющиеся частью пространств имен WPF, но не имеющие префикса x:.

Все расширения разметки реализованы классами, производными от System.Windows.Markup.MarkupExtension. Базовый класс Marku-

pExtension очень прост – он включает единственный метод ProvideValue(), получающий требуемое значение. Другими словами, когда анализатор XAML встречает показанный выше оператор, он создает экземпляр класса StaticExtension, а затем вызывает ProvideValue(), чтобы получить объект, возвращенный статическим свойством SystemColors.ActiveCaption.Brush. Свойство Foreground кнопки cmdAnswer затем устанавливается равным извлеченному объекту.

Конечный результат этого фрагмента XAML-кода эквивалентен тому, как если бы было написано следующее:

```
cmdAnswer.Foreground= SystemColors.ActiveCaptionBrush;
```

Поскольку расширения разметки отображаются на классы, они могут также применяться в виде вложенных свойств. Например, можно использовать StaticExtension со свойством Button.Foreground следующим образом:

```
<Button>
     <Button.Foreground>
     <x:Static Member="SystemColors.ActiveCaptionBrush">
     </x:Static>
</Button.Foreground>
</Button>
```

Как большинство расширений разметки, StaticExtension должен вычисляться во время выполнения, потому что только тогда можно определить текущие системные цвета. Некоторые расширения разметки могут определяться во время компиляции. К ним относятся NullExtension (представляющее значение null) и TypeExtension (конструирующее объект, представляющий тип .NET). Расширения разметки часто используются при привязке данных и при использовании ресурсов.

2.4.4. Прикрепленные свойства

Наряду с обычными свойствами XAML также включает концепцию прикрепленных свойств – свойств, которые могут применяться к

нескольким элементам управления, будучи определенными в другом классе. Эти свойства часто используются для управления компоновкой.

Рассмотрим механизм работы прикрепленных свойств. Каждый элемент управления имеет свой собственный набор внутренних свойств. Например, текстовое поле имеет специфический шрифт, цвет текста и текстовое содержимое – все это определено такими свойствами, как FontFamily, Foreground и Text. При размещении элемента управления внутри контейнера, он получает дополнительные свойства, в зависимости от типа контейнера. Например, если поместить текстовое поле внутрь экранной сетки, то нужно указать ячейку, куда ее помещать. Эти дополнительные детали устанавливаются с использованием прикрепленных свойств.

Прикрепленные свойства всегда используют имя из двух частей в форме ОпределяемыйТип.ИмяСвойства. Этот синтаксис позволяет анализатору XAML различать нормальное свойство и прикрепленное свойство.

В примере с автоответчиком прикрепленные свойства позволяют индивидуальным элементам управления размещать себя в разных строках невидимой сетки:

```
<TextBox …  Grid.Row="0" >
      [Place question here.]
</TextBox>
<Button … Grid.Row="1"
            Ask the Eight Ball
</Button>
<TextBox …
             Grid.Row="2">
      [Answer will appear here.]
</TextBox>
```

Прикрепленные свойства – один из центральныхэлементовWPF. Они действуют как система расширения общего назначения. Напри-

мер, определяя свойство Row как прикрепленное, можно гарантировать его применимость с любым элементом управления.

2.4.5. Специальные символы и пробелы

XAML ограничен правилами XML. Например, XML уделяет особое внимание специальным символам, в частности < и >. Попытка применить эти значения для установки содержимого элемента неизбежно вызовет проблему, поскольку анализатор XAML предположит, что происходит попытка сделать что-то другое – например, создать вложенный элемент.

Предположим, что необходимо создать кнопку, которая содержит текст <Click Me>. Следующий код разметки работать не будет:

```
<Button>
     <ClickMe>
</Button>
```

Проблема состоит в том, что код выглядит так, как будто происходит попытка создать элемент по имени Click с атрибутом Me. Решение состоит в замене сомнительных символов сущностными ссылками – специфическими кодами, которые анализатор XAML интерпретирует правильно. В таблице 1 перечислены специальные символы, которые можно использовать. Отметим, что специальный символ типа кавычки требуется только при установке значений с использованием атрибута, так как кавычка обозначает начало и конец значения атрибута.

Таблица 1

Специальные символы и символы-заменители

Специальный символ	Символ - заменитель
Меньше (<)	<
Больше (>)	>
Амперсанд (&)	&
Кавычка (")	"

Ниже приведен правильный код разметки, использующий соответствующие символьные сущности:

```
<Button>
&lt;Нажмименя&gt;
</Button>
```

Когда анализатор XAML обрабатывает это, он определяет, что необходимо добавить текст <Нажми меня>, и передает строку с этим содержимым, дополняя ее угловыми скобками, свойству Button.Content.

Специальные символы – не единственная тонкость, присутствующая в XAML. Другая проблема – обработка пробелов. По умолчанию XML сокращает все пробелы, а это значит, что длинная строка пробелов, знаков табуляции и жестких переводов строки превращается в единственный пробел. Более того, если добавить пробел перед или после содержимого элемента, этот пробел будет полностью проигнорирован. В некоторых случаях это не то, что нужно. Например, можетвозникнуть необходимость включать серии из нескольких пробелов в текст кнопки. В этом случае следует использовать атрибут xml: space= "preserve" в коде элемента. Пример использования данного атрибута приведен ниже:

```
<TextBoxxml:space="preserve">
            Внутри   данногоTextBox   куча        пробелов
</TextBox>
```

2.4.6. События

До сих пор все атрибуты, которые были показаны выше, отображались на свойства. Однако атрибуты также могут быть использованы для прикрепления обработчиков событий. Синтаксис при этом выглядит следующим образом: ИмяСобытия = «ИмяМетода_Обработчика».

Например, элемент управления Button предоставляет событие Click. Можно прикрепить обработчик событий так, как показано ниже:

```
<ButtonClick="cmdAnswer_Click" </Button>
```

Данное определение предполагает наличие метода по имени cmdAnswer_Click в классе отделенного кода, причем обработчик событий должен иметь правильную сигнатуру. Вот метод, который выполняет этот обработчик:

```
privatevoidcmdAnswer_Click(objectsender, RoutedEventArgse)
{
        this.Cursor = Cursors.Wait;
System.Threading.Thread.Sleep(TimeSpan.FromSeconds(1));
        AnswerGenerator generator = new AnswerGenerator();
        txtAnswer.Text = genera-
tor.GetRandomAnswer(txtQuestion.Text);
        this.Cursor = null;
}
```

Как можно видеть по сигнатуре этого обработчика событий, модель событий в WPF отличается от ранних версий .NET. Она поддерживает новую модель, полагающуюся намаршрутизацию событий. Подробнее эта модель будет рассмотрена ниже.

Во многих ситуациях атрибуты используются для установки свойств и прикрепления обработчиков событий для одного и того же элемента. WPF всегда работает в следующей последовательности: сначала устанавливается свойство Name (если оно есть), затем прикрепляются любые обработчики событий и, наконец, устанавливаются свойства. Это значит, что любые обработчики событий, реагирующие на изменения свойств, будут запущены при первоначальной установке свойства.

В VisualStudio можно использовать средства IntelliSense при добавлении атрибута – обработчика событий. Как только вводится символ равенства (например, после набора Click= в элементе <Button>), он отображает раскрывающийся список со всеми подходящими обработчиками событий в классе отделенного кода, как показано на рис. 4.

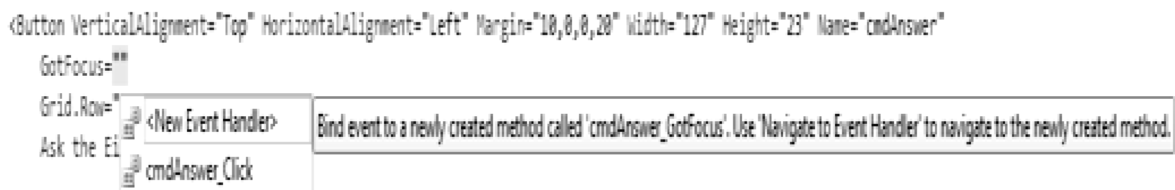

Рис. 4. Прикрепление обработчика события при помощи IntelliSense

Если нужно создать новый обработчик для данного события, достаточно просто выбрать<NewEventHandler> (Новый обработчик событий) в вершине списка.

2.5. Типы из других пространств имен

Выше было показано, как создается базовый интерфейс в XAML с использованием классов, являющихся частью WPF. Однако XAML разрабатывался как средство общего назначения, предназначенное для создания экземпляров объектов .NET, включая те, что находятся в пространствах имен, не относящихся к WPF, и те, которые создаются сторонними разработчиками. Примером этих пространств имен может служить случай, когда необходимо использовать привязку данных и нарисовать информацию из другого объекта, чтобы отобразить ее в элементе управления. Другой пример – когда необходимо установить свойство объекта WPF, используя объект, не относящийся к WPF.

Например, можно заполнить WPF-элемент ListBox объектами данных. ListBox будет вызывать метод ToString(), чтобы получить текст для отображения каждого элемента в списке.

Для того чтобы использовать класс, который не определен ни в одном из пространств имен WPF, следует отобразить пространство имен .NET на пространство имен XML. XAML имеет специальный синтаксис для этого, который выглядит так:xmlns: Префикс= *«clr-namespace:ПространствоИмен; assembly=ИмяСборки»*

Обычно отображение пространства имен помещается в корневой

элемент документа XAML – сразу после атрибутов, которые описывают пространства имен WPF и XAML. Также необходимо заполнить выделенные курсивом части соответствующей информацией, как описано ниже:

- Префикс – префикс XML, который будет использоваться для указания пространства имен в разметке XAML. Например, язык XAML использует префикс x;
- ПространствоИмен – полностью квалифицированное название пространства имен .NET;
- ИмяСборки – сборка, в которой объявлен тип, без расширения .dll. Проект должен ссылаться на эту сборку. Если сборка входит с состав проекта, этот параметр можно опустить.

Например, следующим образом можно получить доступ к базовым типам пространства имен System и отобразить их на префикс sys:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Следующим образом можно получить доступ к типам, которые объявлены в пространстве имен MyProject текущего проекта и отобразить их на префикс local:

xmlns:local="clr-namespace:MyNamespace".

Теперь, чтобы создать экземпляр класса в одном из этих пространств имен, можно использовать префикс пространства имен:

<local:MyObject ...></local:MyObject>.

Следующий пример отображаетотображает префикс sys: на пространство имен System и использует это пространство имен для создания объектов DateTime, которые применяются для заполнения списка:

```
<Window x:Class="WpfApplication2.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525"
xmlns:sys="clr-namespace: System; assembly=mscorlib">
```

```
<ListBox>
<ListBoxItem>
<sys:DateTime>10/13/2010 4:30 PM</sys:DateTime>
</ListBoxItem>
<ListBoxItem>
<sys:DateTime>10/29/2010 12:30 PM</sys:DateTime>
</ListBoxItem>
<ListBoxItem>
<sys:DateTime>10/30/2010 2:30 PM</sys:DateTime>
</ListBoxItem>
</ListBox>
</Window>
```

3. КОМПОНОВКА ЭЛЕМЕНТОВ ИНТЕРФЕЙСА

Значительная часть усилий при проектировании пользовательского интерфейса уходит на организацию содержимого, чтобы она была привлекательной, практичной и гибкой. Но реальная трудность состоит в том, чтобы убедиться в том, что созданная компоновка элементов интерфейса сможет успешно адаптироваться к разным размерам окна.

WPF разделяет компоновку на части, используя разные контейнеры. Каждый контейнер имеет свою собственную логику компоновки – некоторые складывают элементы в стопку, другие распределяют их по ячейкам сетки и т. д. Упор делается на создание более гибких компоновок, которые могут адаптироваться к изменяющемуся содержимому, разным языкам и широкому разнообразию размеров окон.

3.1. Понятие компоновки в WPF

Модель компоновки WPF отражает существенные изменения подхода разработчиков Windows к проектированию пользовательских интерфейсов. Чтобы понять новую модель компоновки WPF, стоит посмотреть на то, что ей предшествовало.

В .NET 1.0 каркас WindowsForms представил весьма примитивную систему компоновки. Элементы управления были фиксированы на месте по жестко закодированным координатам. Единственным удобством былипривязка (anchoring) и стыковка (docking) – два средства, которые позволяли элементам управления перемещаться и изменять свои размеры вместе с их контейнером. Привязка и стыковка были незаменимы для создания простых окон изменяемого размера, например, с привязкой кнопок OK и Cancel (Отмена) к нижнему правому углу окна, либо когда нужно было заставить элемент TreeView разворачиваться для заполнения всей формы. Однако они не могли справиться с более сложными задачами компоновки. Например, привязка и стыковка не позволяли организовать пропорциональное изменение размеров двухпанельных окон, а ткаже в случае высокодинамичного содержимого.

В .NET 2.0 каркас WindowsForms заполнил пробел, благодаря двум новым контейнерам компоновки: FlowLayoutPanel и TableLayoutPanel. Используя эти элементы управления, стало возможным создавать более изощренные интерфейсы. Оба контейнера компоновки позволяли содержащимся в них элементам управления увеличиваться, расталкивая соседние элементы. Это облегчило задачу создания динамического содержимого, создания модульных интерфейсов и локализации приложения.

WPF предлагает систему компоновки, основанную на опыте разработки в WindowsForms. Эта система возвращает модель .NET 2.0, сделав потоковую (flow-based) компоновку стандартной и предоставив лишь рудиментарную поддержку координатной компоновки. Преимущества подобного сдвига огромны. Разработчики могут теперь создавать независящие от разрешения и от размера интерфейсы, которые масштабируются на разных мониторах, автоматически подгоняют себя при изменении содержимого и легко обрабатывают перевод на другие языки.

3.1.1. Базовые принципы компоновки WPF

Окно WPF может содержать только один элемент. Чтобы разместить более одного элемента и создать более практичный пользовательский интерфейс, необходимо поместить в окно контейнер и затем добавлять элементы в этот контейнер.

В WPF компоновка определяется используемым контейнером. Хотя есть несколько контейнеров, среди которых можно выбирать, идеальное окно WPF должно следовать описанным ниже ключевым принципам:

- Элементы не должны иметь явно установленных размеров. Вместо этого они изменяют свой размер, чтобы вместить их содержимое. Например, кнопка увеличивается при добавлении в нее текста. Допустимо ограничить элементы управления приемлемыми размерами, устанавливая максимальное и минимальное их значение;
- Элементы не указывают свою позицию в экранных координатах. Вместо этого они упорядочиваются своим контейнером на основе размера, порядка и другой информации, специфичной для контейнера компоновки. Если необходимо добавить пробел между элементами, то для этого следует использовать свойство Margin;
- Контейнеры компоновки распределяют доступное пространство между своими дочерними элементами. Они пытаются предоставить каждому элементу его предпочтительный размер, если позволяет свободное пространство. Также они могут выделять дополнительное пространство одному или более дочерним элементам;
- Контейнеры компоновки могут быть вложенными. Типичный пользовательский интерфейс начинается с Grid – наиболее развитого контейнера, и содержит другие контейнеры компо-

новки, которые организуют меньшие группы элементов, такие как текстовые поля с метками, элементы списка, пиктограммы в панели инструментов, колонки кнопок и т. д.

Хотя из этих правил существуют исключения, они отражают общие цели проектирования WPF. Т. е., если следовать этим руководствам при построении WPF-приложения, то можно получить лучший, более гибкий пользовательский интерфейс. Если нарушать эти правила, то можно получить пользовательский интерфейс, который не будет хорошо подходить для WPF и его будет значительно сложнее сопровождать.

3.2. Процесс компоновки

Компоновка WPF происходит за две стадии: стадияизмерения и стадиярасстановки. На стадии измерения контейнер выполняет проход в цикле по дочерним элементам и опрашивает их предпочтительные размеры. На стадии расстановки контейнер помещает дочерние элементы в соответствующие им позиции.

Конечно, элемент не может всегда иметь свой предпочтительный размер – иногда контейнер может оказаться недостаточно большим, чтобы его обеспечить. В этом случае контейнер должен усекать такой элемент для того, чтобы вместить его в видимую область. Часто можно избежать такой ситуации, устанавливая минимальный размер окна.

Отметим, что контейнеры компоновки не поддерживают прокрутку. Вместо этого прокрутка обеспечивается специализированным элементом управления содержимым – ScrollViewer, который может быть использован почти где угодно. Подробно данный компонент будет рассмотрен ниже.

3.2.1. Контейнеры компоновки

Все контейнеры компоновки WPF являются панелями, которые унаследованы от абстрактного класса System.Windows.Controls.Panel. Класс Panel добавляет небольшой набор членов, включая три общедоступных свойства, описанные в таблице 2.

Таблица 2

Общедоступные свойства класса Panel

Имя	Описание
Background	Кисть, используемая для рисования фона панели. Необходимо устанавливать это свойство в отличное от null значение, если необходимо принимать события мыши.
Children	Коллекция элементов, находящихся в панели. Это первый уровень элементов.
IsItemsHost	Булевское значение, равное true, если панель используется для показа элементов, ассоциированных с ItemsControl.

Сам по себе базовый класс Panel является начальной точкой для построения других более специализированных классов. WPF предлагает ряд производных от Panel классов, которые можно использовать для организации компоновки. Самые основные классы перечислены в таблице 3. Как и все элементы управления WPF, а также большинство визуальных элементов, эти классы находятся в пространстве имен System.Windows.Controls.

Таблица 3

Основные контейнеры компоновки

Имя	Описание
StackPanel	Размещает элементы в горизонтальный или вертикальный стек. Этот контейнер компоновки обычно используется в небольших секциях крупного более сложного окна.
WarpPanel	Размещает элементы в сериях строк с переносом. В горизонтальной ориентации WrapPanel располагает элементы в строке слева направо, затем переходит к следующей строке. В вертикальной ориентации WrapPanel располагает элементы сверху вниз, используя дополнительные колонки для дополнения оставшихся элементов.
DockPanel	Выравнивает элементы по краю контейнера.
Grid	Выстраивает элементы в строки и колонки невидимой таблицы. Это один из наиболее гибких и широко используемых контейнеров компоновки.
UniformGrid	Помещает элементы в невидимую таблицу, устанавливая

Окончание табл. 3

Имя	Описание
	одинаковый размер для всех ячеек.
Canvas	Позволяет элементам позиционироваться абсолютно – по фиксированным координатам. Эта компоновка неподходящий выбор для окон переменного размера.

Наряду с этими основными контейнерами есть еще несколько более специализированных панелей, которые можно встретить во многих элементах управления. К ним относятся панели, предназначенные для хранения дочерних элементов определенного элемента управления, такого как TabPanel (вкладки в TabControl), ToolbarPanel (кнопки в Toolbar) и ToolbarOverflowPanel (команды в выпадающем меню Toolbar). Имеется еще VirtualizingStackPanel, чей привязанный к данным элемент-список используется для минимизации накладных расходов, а также InkCanvas, который подобен Canvas, но имеет поддержку перьевого ввода на TabletPC.

3.3. Компоновка с помощью StackPanel

StackPanel – один из простейших контейнеров компоновки. Он просто укладывает свои дочерние элементы в одну строку или колонку. Рассмотрим следующее окно, которое содержит стек из четырех кнопок:

```
<Windowx:Class="LayoutPanels.SimpleStack"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="SimpleStack" Height="223" Width="354" MinWidth="50">
<StackPanel Margin="3" Name="stackPanel1">
<Label Margin="3" HorizontalAlignment="Center">
      A Button Stack
</Label>
<Button Margin="3" MaxWidth="200" MinWidth="100">
Button 1</Button>
```

```
<Button Margin="3" MaxWidth="200" MinWidth="100">
Button 2</Button>
<Button Margin="3" MaxWidth="200" MinWidth="100">
Button 3</Button>
<Button Margin="3" MaxWidth="200" MinWidth="100">
Button 4</Button>
<CheckBox Name="chkVertical" Margin="10" HorizontalAlign-
ment="Center"
     Checked="chkVertical_Checked" Un-
checked="chkVertical_Unchecked">
      Use Vertical Orientation</CheckBox>
</StackPanel>
</Window>
```

На рис. 5 показано окно, определяемое данной разметкой.

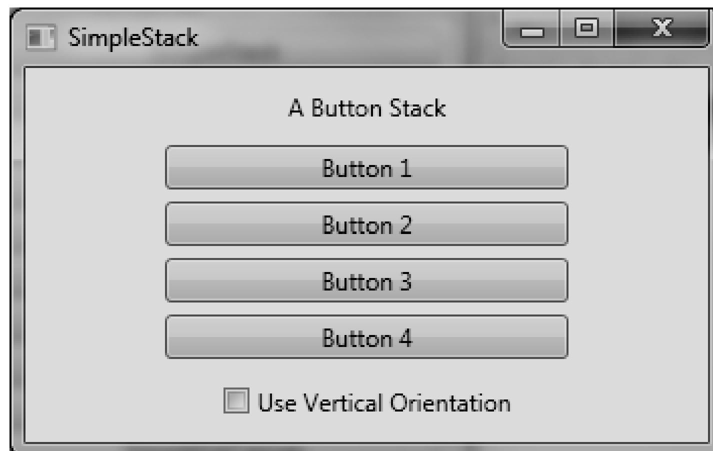

Рис. 5. Простое использование StackPanel

По умолчанию StackPanel располагает элементы сверху вниз, устанавливая высоту каждого из них такой, которая необходима для отображения его содержимого. В данном примере это значит, что размер меток и кнопок устанавливается достаточно большим для комфортабельного размещения текста внутри них. Все элементы растягиваются на полную ширину StackPanel, которая равна ширине окна. Если расширить окно, StackPanel также расширится, и кнопки растянутся, чтобы заполнить ее.

StackPanel может также использоваться для упорядочивания

элементов в горизонтальном направлении посредством установки свойства Orientation:

```
<StackPanelOrientation="Horizontal">
```

Теперь элементы получают свою минимальную ширину (достаточную, чтобы вместить их текст) и растягиваются до полной высоты, чтобы заполнить содержащую их панель. В зависимости от текущего размера окна, это может привести к тому, что некоторые элементы не поместятся, как показано на рис. 6:

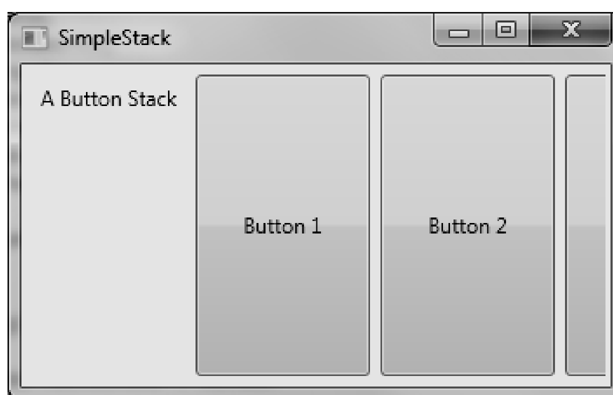

Рис. 6. StackPanel с горизонтальной ориентацией

Очевидно, что этакомпоновка не обеспечивает достаточной гибкости, необходимой реальному приложению. Поэтому можно тонко настраивать способ работы StackPanel и других контейнеров компоновки посредством свойств компоновки, как описано ниже.

3.3.1. Свойства компоновки

Хотя компоновка определяется контейнером, дочерние элементы тоже могут влиять на компоновку. Фактически, панели компоновки взаимодействуют со своими дочерними элементами через небольшой набор свойств компоновки, перечисленных в таблице 4.

Таблица 4

Свойства компоновки

Имя	Описание
HorizontalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по горизонтали. Можно выбрать Center, Left, Right или Stretch.
VerticalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по вертикали. Можно выбрать Center, Top, Bottom или Stretch.
Margin	Добавляет поля вокруг элемента.
MinWidth и MinHeight	Устанавливает минимальные размеры элемента. Если элемент слишком велик, чтобы поместиться в его контейнер компоновки, он будет усечен.
MaxWidth и MaxHeight	Устанавливает максимальные размеры элемента. Если контейнер имеет свободное пространство, элемент не будет увеличен сверх указанных пределов, даже если свойства HorizontalAlignment и VerticalAlignment установлены в Stretch.
Width и Height	Явно устанавливают размеры элемента. Эта установка переопределяет значение Stretch для свойств HorizontalAlignment и VerticalAlignment. Однако этот размер не будет установлен, если выходит за пределы, заданные в MinWidth, MinHeight, MaxWidth и MaxHeight.

Все эти свойства наследуются от базового класса FrameworkElement, и потому поддерживается всеми графическими элементами, которые можно использовать в окне WPF.

3.3.2. Выравнивание

Чтобы понять, как работают эти свойства, еще раз обратим внимание на простуюStackPanel, показанную на рис. 5. В этом примере со StackPanel с вертикальной ориентацией свойство VerticalAlignment не

имеет эффекта, потому что каждый элемент получает такую высоту, которая ему нужна, и не более. Однако свойство HorizontalAlignment имеет значение. Оно определяет место, где располагается каждый элемент в строке.

Обычно HorizontalAlignment по умолчанию равно Left для меток и Stretch – для кнопок. Однако можно изменить эти детали:

```
<StackPanel Margin="3" Name="stackPanel1">
<Label Margin="3" HorizontalAlignment="Center">
      A Button Stack
</Label>
<Button Margin="3" MaxWidth="200" MinWidth="100" HorizontalA-
lignment="Center">Button 1</Button>
<Button Margin="3" MaxWidth="200" MinWidth="100" HorizontalA-
lignment="Left">Button 2</Button>
<Button Margin="3" MaxWidth="200" MinWidth="100" HorizontalA-
lignment="Right">Button 3</Button>
<Button Margin="3" MaxWidth="200" MinWidth="100" HorizontalA-
lignment="Stretch">Button 4</Button>
<CheckBox Name="chkVertical" Margin="10" HorizontalAlign-
ment="Center"
     Checked="chkVertical_Checked" Un-
checked="chkVertical_Unchecked">
      Use Vertical Orientation</CheckBox>
</StackPanel>
```

На рис. 7 показан результат:

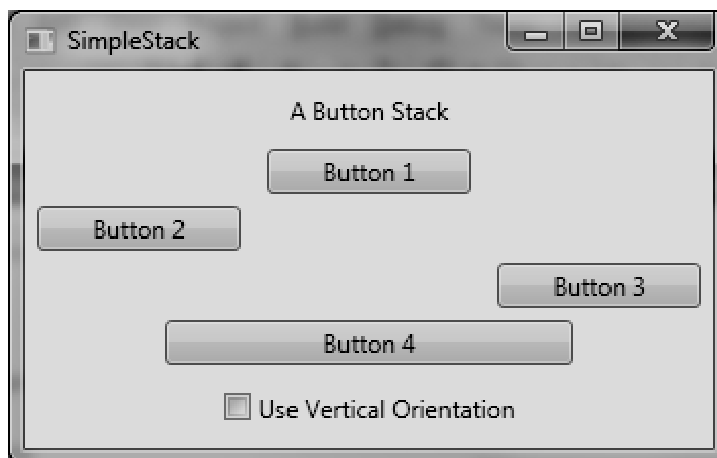

Рис. 7. StackPanel с выровненными кнопками

3.3.3. Поля

Хорошо спроектированное окно должно содержать не только элементы. Оно также содержит немного дополнительного пространства между элементами. Чтобы представить это дополнительное пространство и сделать пример StackPanel менее «зажатым», можно устанавливать поля вокруг элементов управления.

При установке полей можно установить одинаковую ширину для всех сторон, как в рассмотренных примерах:

```
<ButtonMargin="3">Button 4</Button>
```

Также можно установить разные поля для каждой стороны элемента управления в следующем порядке: левое, верхнее, правое, нижнее:

```
<ButtonMargin="5,10,5,10">Button 1</Button>
```

В коде поля устанавливаются при помощи структуры Thickness:

```
cmd.Margin= newThickness(5);
```

Определение правильных полей вокруг элементов управления – отчасти искусство, потому что необходимо учитывать, какимобразом установки полей соседних элементов управления влияют друг на друга.

В идеале следует сохранять разные установки полей насколько возможно согласованными и избегать разных значений для полей разных сторон. Например, в примере со StackPanel имеет смысл использовать одинаковые поля для кнопок и самой панели, как было показано в примерах выше. Таким образом, общее пространство между двумя кнопками (сумма полей двух кнопок) получается таким же, как общее пространство между кнопкой и краем окна (сумма поля кнопки и поля StackPanel).

3.3.4. Минимальный, максимальный и явный размеры

И, наконец, каждый элемент включает свойства Height и Width, которые позволяют установить явный размер. Однако предпринимать такой шаг – идея, противоречащая правилам WPF.В хорошо спроектированной компоновке в этом не должно быть необходимости. Если

добавляется информация о размерах, то можно создать хрупкую компоновку, которая не может адаптироваться к изменениям (таким как разные языки и размеры окон) и усекает содержимое.

Вместо этого при необходимости используйте свойства минимального и максимального размеров, чтобы зафиксировать элемент управления в нужных пределах размеров.

Следует обратить внимание на то, что в примерах, приведенных выше, для кнопок заданы максимальный и минимальный размеры.

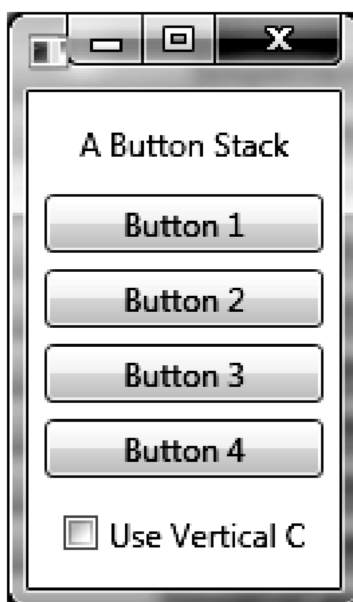

Рис. 8. StackPanel в минимальном размере

Нарис. 8 представлено окно в минимальном размере. Кнопки имеют размер по 100 единиц каждая, и окно не может быть сужено, чтобы сделать их меньше. Попытка сжать окно от этой точки приведет к тому, что правая часть каждой кнопки будет усечена. При увеличении окна кнопки также растут, пока не достигнут своего максимума в 200 единиц. Если после этого увеличение окна продолжится, то с каждой стороны от кнопок будет добавлено дополнительное пространство (как показано на рис. 5).

В некоторых ситуациях можно использовать код, проверяющий, насколько велик элемент в окне. Свойства Height и Width в данном случае не актуальны, потому что указывают желаемые установки

размера, которые могут не соответствовать действительному визуализируемому размеру. Их действительный размер, используемый для визуализации элемента, можно узнать, прочитав свойства ActualHeight и ActualWidth. Следует помнить, что эти значения могут меняться при изменении размера окна или содержимого элементов.

3.3.5. Окна с автоматическими размерами

В рассматриваемом примере есть один элемент с жестко закодированным размером: окно верхнего уровня, котороесодержит в себе StackPanel (и всем прочим внутри). По ряду причин жестко кодировать размеры окна имеет смысл:

- Во многих случаях может возникнуть необходимость сделать окно меньше, чем диктует желаемый размер его дочерних элементов. Например, если окно включает контейнер с прокручиваемым текстом, можетвозникнуть необходимость ограничить размер этого контейнера, чтобы прокрутка была возможна;
- Минимальный размер окна может быть удобен, но при этом не иметь наиболее привлекательных пропорций;
- Автоматическое изменение размеров окна не ограничено размером дисплея монитора. Поэтому окно с автоматически установленным размером может оказаться слишком большим для просмотра.

Однако окна с автоматически устанавливаемым размером возможны, и они имеют смысл, если создается простое окно с динамическим содержимым. Чтобы включить автоматическую установку размеров окна нужно удалить свойства Height и Width и установить Window.SizeToContent равным WidthAndHeight. Окно сделает себя достаточно большим, чтобы вместить его содержимое. Также можно позволить окну изменять свой размер только в одном измерении, используя значение SizeToContent для Width или Height.

3.4. WrapPanel и DockPanel

Очевидно, что только StackPanel не может помочь в создании реалистичного пользовательского интерфейса. Для этого StackPanel должен работать с другими более развитыми контейнерами компоновки.

Наиболее изощренный контейнер компоновки – это Grid, который будет рассмотрен далее в этомразделе. Но сначала стоит рассмотреть WrapPanel и DockPanel – два самыхпростых контейнера компоновки, предоставленных WPF. Они дополняют StackPanel другим поведением компоновки.

3.4.1. WrapPanel

WrapPanel располагает элементы управления в доступном пространстве – по одной строке или колонке за раз. По умолчанию свойство WrapPanel.Orientation устанавливается в Horizontal; элементы управления располагаются слева направо, затем – в следующих строках. Однако можно использовать Vertical для размещения элементов в нескольких колонках. Подобно StackPanel, панель WrapPanel предназначена для управления мелкими деталями пользовательского интерфейса, а не компоновкой всего окна. Например, можно использовать WrapPanel для удержания вместе кнопок в элементе управления типа панели инструментов.

Приведем пример, определяющий серии кнопок с разными выравниваниями и помещением их в WrapPanel:

```
<WrapPanel Margin="3">
<Button VerticalAlignment="Top">Top Button</Button>
<Button MinHeight="60">Tall Button 2</Button>
<Button VerticalAlignment="Bottom">Bottom Button</Button>
<Button>Stretch Button</Button>
<Button VerticalAlignment="Center">Centered Button</Button>
</WrapPanel>
```

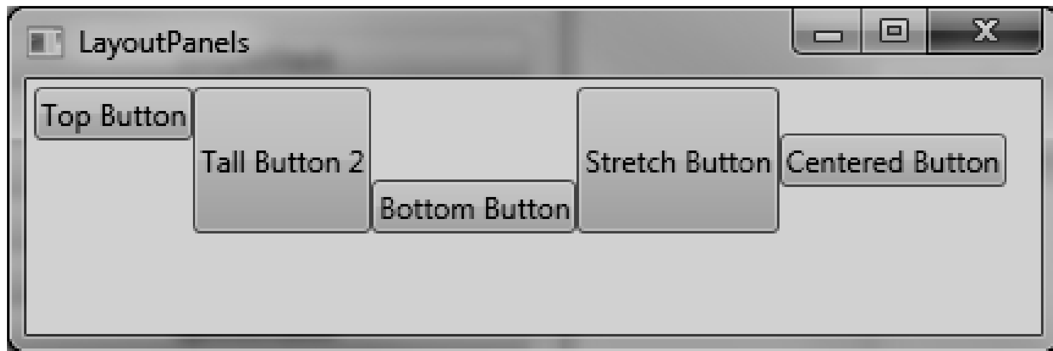

Рис. 9. Пример использования WrapPanel

На рис. 9 показано, как переносятся кнопки, чтобы заполнить текущий размер WrapPanel, определяемый размером содержащего его окна. Как демонстрирует этот пример, WrapPanel в горизонтальном режиме создает серии воображаемых строк, высота каждой из которых определяется высотой самого крупного содержащегося в ней элемента. Другие элементы управления могут быть растянуты для заполнения строки или выровнены в соответствии со значением свойства VerticalAlignment.

3.4.2. DocPanel

DockPanel обеспечивает иной вариант компоновки. Эта панель растягивает элементы управления вдоль одной из внешних границ. Простейший способ визуализации компоновки – представить линейку инструментов, которая присутствует в верхней части многих приложений Windows. Такие линейки инструментов прикрепляются к вершине окна. Как и в случае StackPanel, прикрепленные элементы должны выбрать один аспект компоновки. Например, если прикрепить кнопку к вершине DockPanel, она растянется на всю ширину DockPanel, но получит высоту, которая ей потребуется. С другой стороны, если прикрепить кнопку к левой стороне контейнера, ее высота будет растянута для заполнения контейнера, но ширина будет установлена по необходимости.

Сторона, к которой будут стыковаться элементы, устанавливается через прикрепленное свойство по имени Dock, которое может

быть установлено в Left, Right, Top или Bottom. Каждый элемент, помещаемый внутри DockPanel, автоматически получает это свойство.

Ниже приведен пример использования DockPanel:

```
<DockPanel LastChildFill="True">
<Button DockPanel.Dock="Top">A Stretched Top Button</Button>
<Button DockPanel.Dock="Top" HorizontalAlignment="Center">A
Centered Top Button</Button>
<Button DockPanel.Dock="Top" HorizontalAlignment="Left">A Left-
Aligned Top Button</Button>
<Button DockPanel.Dock="Bottom">Bottom Button</Button>
<Button DockPanel.Dock="Left">Left Button</Button>
<Button DockPanel.Dock="Right">Right Button</Button>
<Button >Remaining Space</Button>
</DockPanel>
```

В этом примере также устанавливается LastChildFill в true, что указывает DockPanel о необходимости отдать оставшееся пространство последнему элементу. Результат показан на рис. 10.

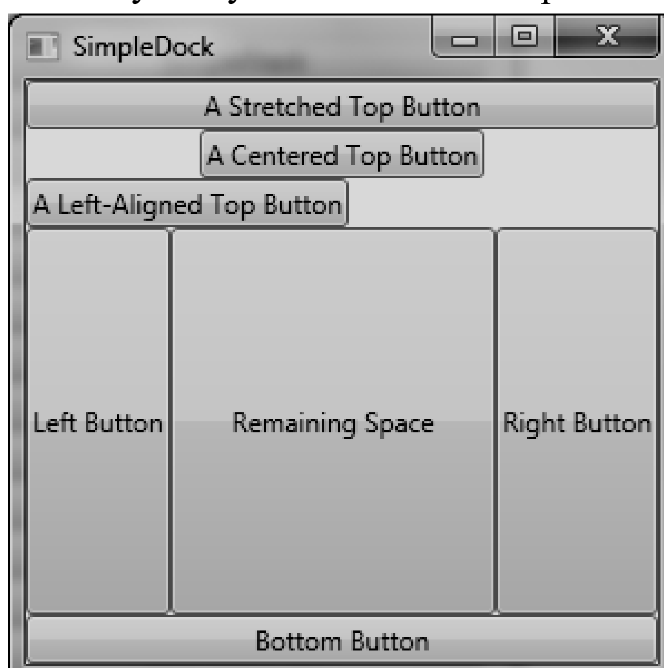

Рис. 10. Пример использования DockPanel

Очевидно, что при такой стыковке элементов управления важен порядок. В данном примере верхняя и нижняя кнопки получают всю ширину DockPanel, поскольку они стыкованы первыми. Когда затем

стыкуются левая и правая кнопки, они помещаются между первыми двумя. Если поступить наоборот, то левая и правая кнопки получат полную высоту сторон панели, а верхняя и нижняя станут уже, потому что им придется размещаться уже между боковыми кнопками.

Можно стыковать несколько элементов к одной стороне. В этом случае элементы просто выстраиваются вдоль этой стороны в том порядке, в котором они объявлены в разметке.

3.4.3. Вложение контейнеров компоновки

StackPanel, WrapPanel и DockPanel редко используются сами по себе. Вместо этого они применяются для формирования частей интерфейса. Например, можно использовать DockPanel для размещения разных контейнеров StackPanel и WrapPanel в соответствующих областях окна.

Например, предположим, что необходимо создать стандартное диалоговое окно с кнопками ОК и Cancel (Отмена) в нижнем правом углу, расположив большую область содержимого в остальной части окна. Есть несколько способов смоделировать этот интерфейс в WPF, но простейший вариант, использующий панели, рассмотренные выше, выглядит следующим образом:

1. Создать горизонтальную StackPanel для размещения рядом кнопок ОК и Cancel;
2. Поместить StackPanel в DockPanel и использовать ее для стыковки к нижней части окна;
3. Установить DockPanel. LastChildFill в true, чтобы можно было использовать остаток окна для заполнения прочим содержимым;
4. Установить значения полей, чтобы распределить пустое пространство.

Вот как выглядит итоговая разметка:

```
<DockPanel LastChildFill="True">
<StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Right"
Orientation="Horizontal">
```

```
<Button Margin="10,10,2,10" Padding="3,3,3,3">OK</Button>
<Button Margin="2,10,10,10" Padding="3,3,3,3">Cancel</Button>
</StackPanel>
<TextBox DockPanel.Dock="Top" Mar-
gin="10">Авотипроверочкапришла.</TextBox>
</DockPanel>
```

На рис. 11 показано диалоговое окно, определяемое приведенной разметкой.

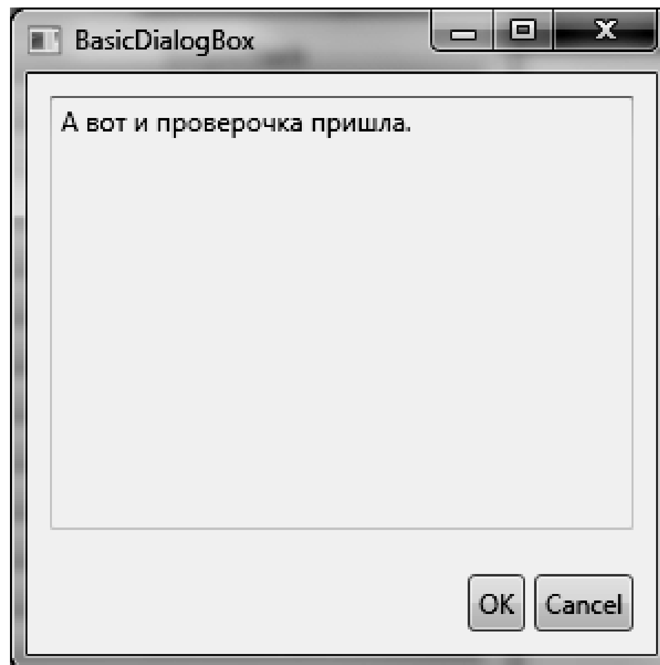

Рис. 11. Пример диалогового окна

На первый взгляд может показаться, что все это требует больше работы, чем точное размещение с использованием координат в традиционном приложении Windows Forms. Во многих случаях это действительно так. Однако большие затраты времени на разработку компоновки компенсируются легкостью, с которой будущем можно будет изменять пользовательский интерфейс. Например, если будет принято решение поместить кнопки OK и Cancel в центре нижней части окна, достаточно просто изменить выравнивание содержащей их StackPanel:

```
<StackPanel … HorizontalAlignment="Center" …>
```

В данном примере структура разметки сравнительно проста, однако, если имеется плотное дерево элементов, легко потерять пред-

ставление об общей структуре. Visual Studio предлагает удобное средство, формирующее древовидное представление элементов интерфейса. Такое представление позволяет легко выбирать элемент, который нужно увидеть или модифицировать. Это средство – окно Document Outline (Эскиз документа), которое можно отобразить, выбрав View – Other Windows – Document Outline (Вид – Другие окна – Эскиз документа) из главного меню. Само окно представлено на рис. 12.

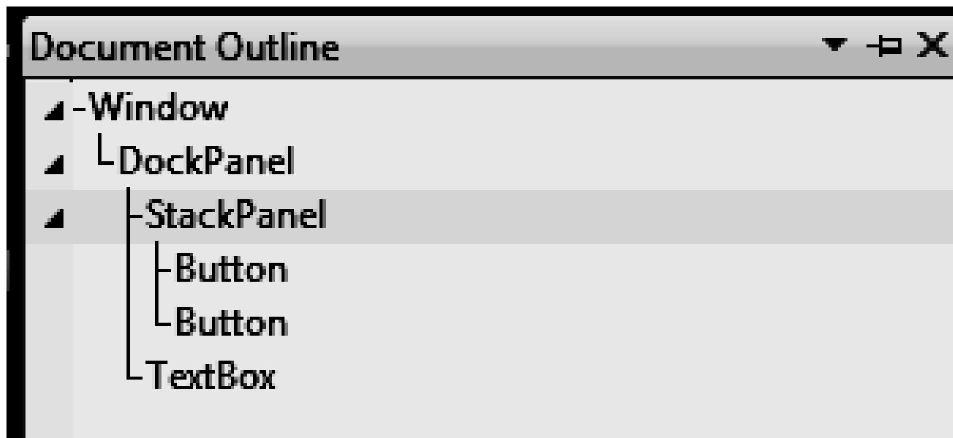

Рис. 12. Окно Document Outline

3.5. Grid

Grid – наиболее мощный контейнер компоновки в WPF. Большая часть того, что можно достичь с другими элементами управления компоновкой, также возможно и в Grid. Контейнер Grid является идеальным инструментом для разбиения окна на меньшие области, которыми можно управлять с помощью других панелей. Фактически Grid настолько удобен, что при добавлении нового документа XAML для окна в среде разработки она автоматически добавляет дескрипторы Grid в качестве контейнера первого уровня, вложенного внутрь корневого элемента Window.

Grid располагает элементы в невидимой сетке строк и колонок. Хотя в отдельную ячейку этой сетки можно поместить более одного элемента, обычно имеет смысл помещать в каждую ячейку по одному элементу. Очевидно, что этот элемент сам может быть другим контейнером компоновки, который организует свою собственную группу содержащихся в нем элементов управления.

Хотя Grid задуман как невидимый, можно установить свойство Grid.ShowGridLines в true, чтобы получить наглядное представление о нем. Это средство помогает облегчить отладку за счет наглядной демонстрации разбиения пространства Grid на отдельные области. Это важно, поскольку появляется возможность точно контролировать то, как Grid выбирает ширину колонок и высоту строк.

Создание компоновки на основе Grid выполняется в два этапа. Сначала выбирается нужное количество колонок и строк. Затем каждому содержащемуся элементу назначается соответствующая строка и колонка, тем самым помещая его в правильное место.

Колонки и строки создаются путем заполнения объектами коллекции Grid.ColumnDefinitions и Grid.RowDefinitions. Например, если необходимо создать две строки и три колонки, нужно добавить следующие дескрипторы:

```
<Grid ShowGridLines="True">
<Grid.RowDefinitions>
<RowDefinition></RowDefinition>
<RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition></ColumnDefinition>
<ColumnDefinition></ColumnDefinition>
<ColumnDefinition></ColumnDefinition>
</Grid.ColumnDefinitions>
</Grid>
```

Как демонстрирует этот пример, не обязательно указывать какую-либо информацию в элементах RowDefinition или ColumnDefinition. Если оставить их пустыми (как в показанном примере), то Grid поровну разделит пространство между всеми строками и колонками. В данном примере каждая ячейка будет одного и того же размера, который зависит от размера включающего окна.

Чтобы поместить индивидуальные элементы в конкретную

ячейку, нужно использовать прикрепленные свойства Row и Column. Оба эти свойства принимают числовое значение индекса, начиная с 0. Например, вот как можно создать частично заполненную кнопками сетку:

```
<Grid ShowGridLines="True">
    …
<Button Grid.Row="0" Grid.Column="0">Top Left</Button>
<Button Grid.Row="0" Grid.Column="1">Middle Left</Button>
<Button Grid.Row="1" Grid.Column="2">Bottom Right</Button>
<Button Grid.Row="1" Grid.Column="1">Bottom Middle</Button>
</Grid>
```

Каждый элемент должен быть помещен в свою ячейку явно. Это позволяет помещать в одну ячейку более одного элемента, или же оставлять определенные ячейки пустыми (что часто бывает полезным). Это также означает, что можно объявлять элементы интерфейса в любом порядке. Однако общепринятым порядком определения является построчное определение элементов управления, а в каждой строке – определение слева направо. Если явно не указать свойство Grid.Row, то Grid предполагает его равным 0. То же самое касается и свойства Grid.Column. Таким образом, если пропустить оба атрибута элемента, он помещается в первую ячейку Grid.

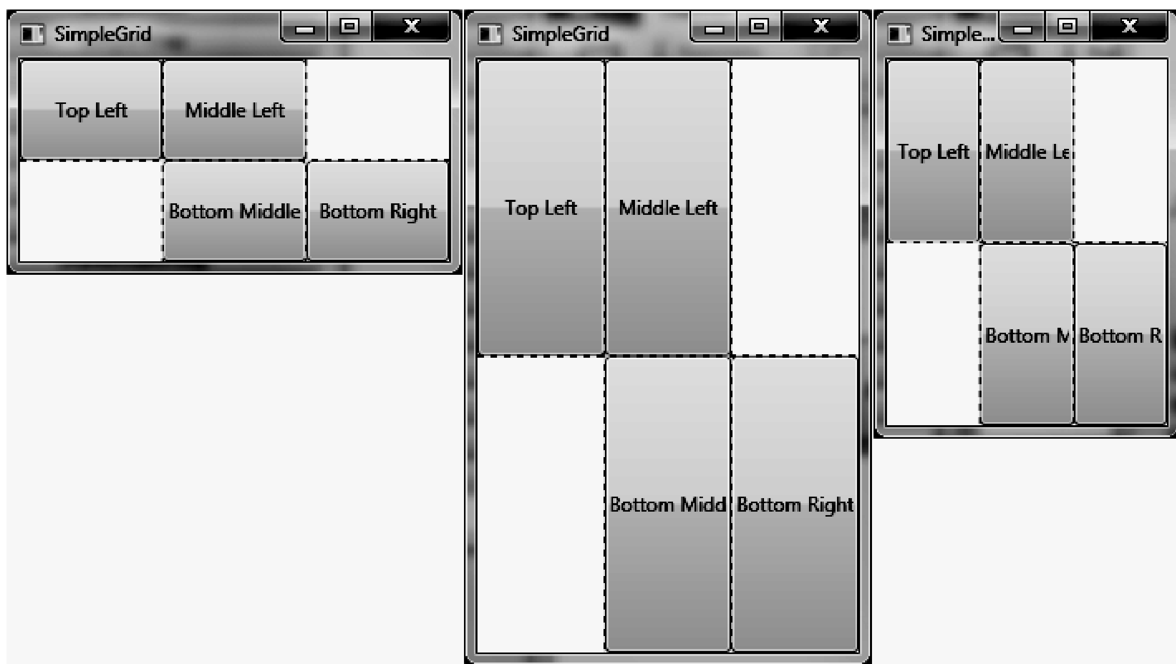

Рис. 12. Отображение простой сетки при различных размерах окна

На рис. 13 показано, как эта простая сетка выглядит при разных размерах окна. Отметим также, что свойство ShowGridLines установлено в true, поэтому можно видеть границы между колонками и строками.

Grid предоставляет базовый набор свойств компоновки, перечисленных в таблице 4. Это значит, что можно добавлять поля вокруг содержимого ячейки, можно менять режим изменения размера, чтобы элемент не рос, заполняя ячейку целиком, а также можно выравнивать элемент по одному из граней ячейки. Если элемент будет иметь размер, превышающий тот, что может вместить ячейка (см. Рис. 12, правая часть), часть содержимого будет отсечена.

3.5.1. Использование Grid в Visual Studio

При использовании Grid при проектировании в Visual Studio, можно отметить, что он работает несколько иначе, чем другие контейнеры компоновки. При перетаскивании элемента на Grid Visual Studio позволяет поместить его в точную позицию. Visual Studio выполняет такое размещение, устанавливая свойство Margin элемента.

При установке полей Visual Studio использует ближайший угол. Например, если элемент – ближайший к верхнему левому углу Grid, то Visual Studio устанавливает верхнее и левое поля для позиционирования элемента (оставляя правое и нижнее поля равными 0). При перетаскивании элемента ниже, приближая его к нижнему левому углу, то Visual Studio устанавливает вместо этого нижнее и левое поля и устанавливает свойство Vertical Alignment в Bottom. Это очевидно влияет на то, как перемещается элемент при изменении размера Grid. Процесс установки полей в Visual Studio выглядит достаточно прямолинейным, но в большинстве случаев он приводит не к тому результату, который нужен. Обычно необходима более гибкая потоковая (flow) компоновка, которая позволяет некоторым элементам расширяться динамически, изменяя местоположение соседей. В этом случае жесткое кодирование позиции свойством Margin является со-

вершенно негибким. Проблема усугубляется, когда происходит добавление множества элементов, потому что Visual Studio не добавляет автоматически новых ячеек. В результате все добавленные элементы помещаются в одну и ту же ячейку. Разные элементы могут выравниваться по разным углам Grid, что заставит их перемещаться относительно друг друга (и даже перекрывать друг друга) при изменении размера окна.

Однажды поняв, как работает Grid, можно исправить эти проблемы. Один изспособов решения заключается в конфигурировании Grid перед началом добавления элементов посредством определения новых строк и колонок. Однажды настроив Grid, можно перетаскивать в него нужные элементы и конфигурировать их настройки полей и выравнивание в окне Properties, или редактируя XAML вручную.

3.5.2. Тонкая настройка строк и колонок

Если бы Grid был просто коллекцией строк и колонок пропорциональных размеров, от него было бы мало толку. Поэтому Grid поддерживает способы изменять размеры каждой строки и колонки.

Grid поддерживает следующие стратегии изменения размеров:

- Абсолютные размеры. В данном варианте выбирается точный размер, используя независимые от устройства единицы измерения. Это наименее удобная стратегия, поскольку она недостаточно гибка, чтобы справиться с изменением размеров содержимого, изменением размеров контейнера или локализацией;
- Автоматические размеры. Каждая строка и колонка получает в точности то пространство, которое нужно, и не более. Это один из наиболее удобных режимов изменения размеров;
- Пропорциональные размеры. Пространство разделяется между группой строк и колонок. Это стандартная установка для всех строк и колонок. Например, на Рис. 12 можно видеть, что все ячейки увеличиваются пропорционально при расширении Grid.

Для максимальной гибкости можно смешивать и сочетать эти разные режимы изменения размеров. Например, часто удобно создать несколько автоматически изменяющих размер строк и затем позволить одной или двум остальным строкам поделить между собой оставшееся пространство через пропорциональную установку размеров.

Режим изменения размеров устанавливается, используя свойство Width объекта ColumnDefinition или свойство Height объекта RowDefinition, присваивая им некоторое число или строку. Например, в следующем примере устанавливается абсолютная ширина в 100 независимых от устройства единиц:

```
<ColumnDefinitionWidth="100"></ColumnDefinition>
```

Чтобы использовать пропорциональное изменение размеров, указывается значение Auto:

```
<ColumnDefinitionWidth="Auto"></ColumnDefinition>
```

И, наконец, чтобы использовать пропорциональное изменение размеров, задается звездочка (*):

```
<ColumnDefinitionWidth="*"></ColumnDefinition>
```

Если используется смесь пропорциональной установки размеров с другими режимами, то пропорционально изменяемая строка или колонка получит все оставшееся пространство.

Если необходимо разделить оставшееся пространство неравными частями, можно присвоить вес, который следует поместить перед звездочкой. Например, если есть две строки пропорционального размера, и нужно, чтобы высота первой была равна половине высоты второй, можно разделить оставшееся пространство следующим образом:

```
<RowDefinition Height="*"></RowDefinition>
<RowDefinition Height="2*"></RowDefinition>
```

Это сообщит Grid о том, что высота второй строки должна быть вдвое больше высоты первой строки. Можно указывать любые числа, чтобы делить дополнительное пространство.

Продублируем пример диалогового окна, показанного на

рис. 11, используя эти режимы установки размеров. Используем контейнер Grid верхнего уровня для разделения окна на две строки вместо использования DockPanel. Для этого будет использоваться следующая разметка:

```
<GridShowGridLines="True">
<Grid.RowDefinitions>
<RowDefinition Height="*"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
</Grid.RowDefinitions>
<TextBox Margin="10" Grid.Row="0">This is a test.
</TextBox>
<StackPanel Grid.Row="1" HorizontalAlignment="Right" Orienta-
tion="Horizontal">
<Button Margin="10,10,2,10"  Padding="3">OK</Button>
<Button Margin="2,10,10,10" Padding="3">Cancel</Button>
</StackPanel>
</Grid>
```

Этот код разметки немного длиннее первоначального, но он имеет то преимущество, что объявляет элементы управления в порядке их появления, что облегчает его понимание. В этом случае выбор такого подхода – просто вопрос предпочтений. При желании можно заменить его вложенным StackPanel с однострочным и одноколоночным Grid.

Отметим, что практически любой интерфейс можно создать, используя вложенные контейнеры Grid. Однако, если происходит работа с небольшими разделами пользовательского интерфейса или расположением небольшого количества элементов, то часто проще применить более специализированные контейнеры StackPanel и DockPanel.

3.5.3. Объединение строк и колонок

Выше было продемонстрировано, как помещаются элементы в ячейки с использованием прикрепленных свойств Row и Column.

Также можно использовать еще два прикрепленных свойства, чтобы растянуть элемент на несколько ячеек: RowSpan и ColumnSpan. Эти свойства принимают количество строк или колонок, которые должен занять элемент.

Например, следующая кнопка займет все место, доступное в первой и второй ячейках первой строки:

```
<Button Grid.Row="0" Grid.Column="0" Grid.RowSpan="2">Span But-
ton</Button>
```

А эта кнопка растянется всего на четыре ячейки, охватив две колонки и две строки:

```
<Button Grid.Row="0" Grid.Column="0" Grid.RowSpan="2"
Grid.ColumnSpan="2">Span Button</Button>
```

Объединение нескольких строк и колонок позволяет достичь некоторых интересных эффектов, и особенно удобно, если необходимо вместить в табличную структуру элементы, которые меньше или больше имеющихся ячеек.

Используя объединение колонок, можно переписать пример простого диалогового окна на рис. 11, используя при этом единственный Grid. Этот Grid делит окно на три колонки, растягивая текстовое поле на все три, и использует последние две колонки для выравнивания кнопок ОК и Cancel (Отмена).

```
<Grid ShowGridLines="True">
<Grid.RowDefinitions>
<RowDefinition Height="*"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="*"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
<TextBox Margin="10" Grid.Row="0" Grid.Column="0"
```

```
Grid.ColumnSpan="3">Авотипроверкаприехала
</TextBox>
<Button Margin="10,10,2,10" Padding="3" Grid.Row="1"
Grid.Column="1">OK</Button>
<Button Margin="2,10,10,10" Padding="3" Grid.Row="1"
Grid.Column="2">Cancel</Button>
</Grid>
```

Очевидно, что такая компоновка неясна и непонятна. Ширины колонок определяются размером двух кнопок окна, что затрудняет добавление нового содержимого к существующей структуре Grid. Если возникнет необходимость сделать даже минимальное дополнение к этому окну, вероятно, будет нужно создать для этого новый набор колонок.

Таким образом, при выборе для окна контейнера компоновки, не просто нужно добиться корректного поведения компоновки. Также нужно построить структуру компоновки, которую легко сопровождать и расширять в будущем. Хорошее эмпирическое правило заключается в использовании меньших контейнеров компоновки, подобных StackPanel для одноразовых задач компоновки, таких как организация группы кнопок. С другой стороны, если нужно применить согласованную структуру более чем к одной области окна (как с колонкой текстового поля, показанной ниже, на рис. 14), то в этом случае Grid – незаменимый инструмент для стандартизации компоновки.

3.5.4. Разделенные окна

Каждый пользователь Windows видел разделительные полосы – перемещаемые разделители, отделяющие одну часть окна от другой. Например, в проводнике Windows слева располагается список папок, а справа – список файлов. Можно перетаскивать разделительную полосу, устанавливая пропорции между этими двумя панелями в окне.

В WPF полосы разделителей представлены классом GridSplitter и являются средствами Grid. Добавляя GridSplitter к Grid, можно предоставить пользователю возможность изменения размеров строк и

колонок. На рис. 14 показано окно, в котором GridSplitter находится между двумя колонками. Перетаскивая полосу разделителя, пользователь может менять относительные ширины обеих колонок.

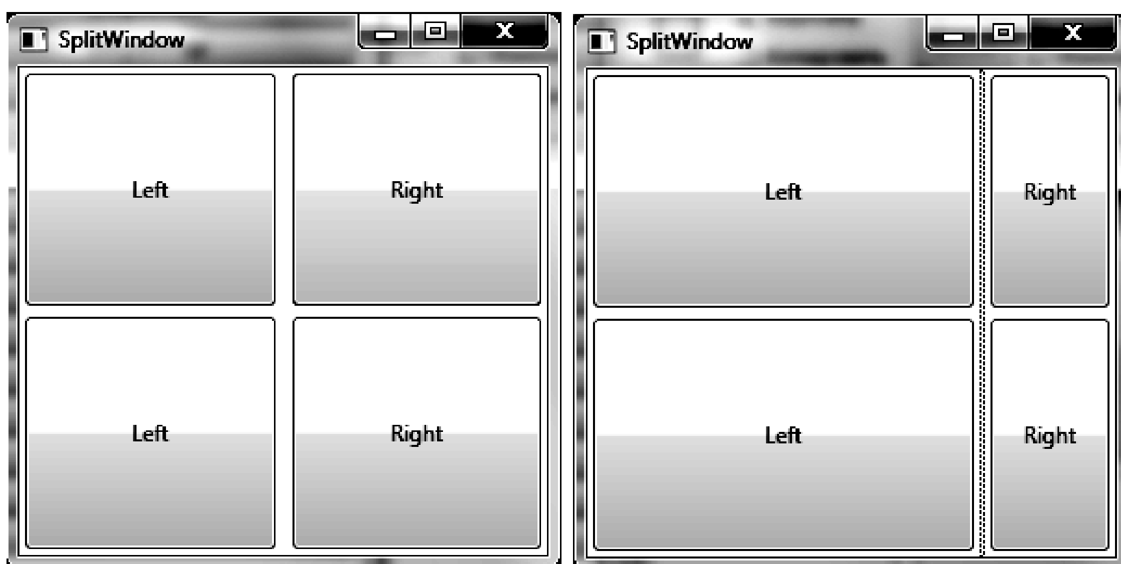

Рис. 13. Перемещение полосы разделителя

Использование класса GridSplitterподчиняется некоторым правилам:

- GridSplitter должен быть помещен в ячейку Grid. Можно поместить GridSplitter в ячейку с существующим содержимым – тогда следует настроить установки полей, чтобы они не перекрывались. Лучший подход заключается в резервировании специальной колонки или строки для GridSplitter, со значениями Height или Width, равными Auto;
- GridSplitter всегда изменяет размер всей строки или колонки, а не отдельной ячейки. Чтобы сделать внешний вид GridSplitter соответствующим такому поведению, необходимо растянуть GridSplitter по всей строке или колонке, а не ограничиваться единственной ячейкой. Чтобы достичь этого, можно использовать свойства RowSpan или ColumnSpan, которые были рассмотрены выше. Например, GridSplitter на рис. 14 имеет значение RowSpan, равное 2. В результате он растягивается на всю колонку. Если не добавить эту установку, он появится только в верхней строке, где он помещен, даже несмотря на

то, что перемещение разделительной полосы изменило бы размер всей колонки;

- Изначально GridSplitter настолько мал, что его не видно. Чтобы сделать его удобным, нужно задать его минимальный размер. В случае вертикальной разделяющей полосы нужно установить VerticalAlignment в Stretch, aWidth – в фиксированный размер (например, в 10 единиц). В случае горизонтальной разделительной полосы следует установить HorizontalAlignment в Stretch, а Height – в фиксированный размер;
- Выравнивание GridSplitter также определяет, будет ли разделительная полоса горизонтальной или вертикальной. В случае горизонтальной разделительной полосы следует установить VerticalAlignment в Center (что является значением по умолчанию), указав тем самым, что перетаскивание разделителя изменит размеры строк, находящихся выше и ниже. В случае вертикальной разделительной полосы следует установить HorizontalAlignment в Center, чтобы изменять размеры соседних колонок.

Код разметки примера, показанного на рис. 14, приведен ниже, а детали GridSplitter выделены полужирным курсивом:

```
<Grid>
<Grid.RowDefinitions>
<RowDefinition></RowDefinition>
<RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition MinWidth="100"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition MinWidth="50"></ColumnDefinition>
</Grid.ColumnDefinitions>
```

```
<Button Grid.Row="0" Grid.Column="0" Margin="3">Left</Button>
<Button Grid.Row="0" Grid.Column="2" Margin="3">Right</Button>
<Button Grid.Row="1" Grid.Column="0" Margin="3">Left</Button>
<Button Grid.Row="1" Grid.Column="2" Margin="3">Right</Button>
<GridSplitter Grid.Row="0" Grid.Column="1" Grid.RowSpan="2"
 Width="3" VerticalAlignment="Stretch" HorizontalAlign-
ment="Center"ShowsPreview="False"></GridSplitter>
</Grid>
```

Эта разметка включает одну дополнительную деталь. Когда объявляется GridSplitter, свойство ShowPreview устанавливается в false. В результате при перетаскивании полосы разделителя от одной стороны к другой колонки изменяют свой размер немедленно. Если установить ShowPreview в true, то при перетаскивании будет отображаться лишь серая тень, следующую за курсором мыши, которая покажет, где разделитель окажется после того, как кнопка мыши будет отпущена. Колонки не изменят своего размера вплоть до этого момента. Можно также использовать клавиши со стрелками для изменения размера GridSplitter после того, как он получит фокус ввода.

Также можно изменить свойство DragIncrement, если необходимо заставить полосу разделителя перемещаться дискретными шагами (например, по 10 единиц за раз). Если есть необходимость контролировать минимальный и максимальный допустимые размеры колонок, достаточно просто установить соответствующие свойства в разделе ColumnDefinitions.

Обычно Grid содержит не более одного GridSplitter. Однако можно вкладывать один Grid в другой, и при этом каждый из них будет иметь собственный GridSplitter. Это позволит создавать окна, которые разделены на две области (например, на левую и правую панель), одна из которых (например, правая), в свою очередь, также будет разделена на два раздела с изменяемыми размерами. Ниже приведен пример такой разметки:

```
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition></ColumnDefinition>
</Grid.ColumnDefinitions>
<Grid Grid.Column="0" VerticalAlignment="Stretch">
<Grid.RowDefinitions>
<RowDefinition></RowDefinition>
<RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Button Margin="3" Grid.Row="0">Top Left</Button>
<Button Margin="3" Grid.Row="1">Bottom Left</Button>
</Grid>
<GridSplitter Grid.Column="1" Width="3"
VerticalAlignment="Stretch" HorizontalAlignment="Center"
ShowsPreview="False"></GridSplitter>
<Grid Grid.Column="2">
<Grid.RowDefinitions>
<RowDefinition></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Button Grid.Row="0" Margin="3">Top Right</Button>
<Button Grid.Row="2" Margin="3">Bottom Right</Button>
<GridSplitter Grid.Row="1"
                  Height="3" VerticalAlignment="Center"
                  HorizontalAlignment="Stretch"
                  ShowsPreview="False"></GridSplitter>
</Grid>
</Grid>
```

Окно, определяемое данной разметкой, показано на рис. 15.

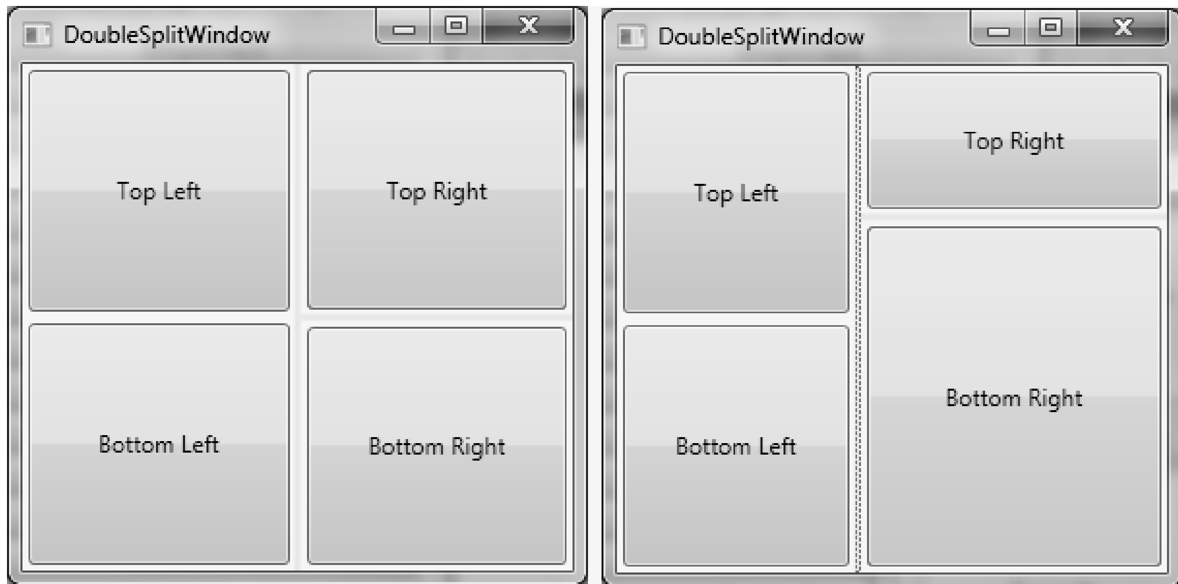

Рис. 14. Окно с двумя разделителями

3.5.5. Группы с общими размерами

Как было показано выше, Grid содержит коллекцию строк и колонок, размер которых устанавливается явно, пропорционально или на основе размеров их дочерних элементов. Есть только один способ изменить размер строки или колонки – приравнять его к размеру другой строки или колонки. Это выполняется при помощи средства, называемого группами с общими размерами.

Цель таких групп – поддержание согласованности между различными частями пользовательского интерфейса. Например, можно установить размер одной колонки в соответствии с ее содержимым, а размер другой колонки – в точности равным размеру первой. Однако реальное преимущество групп с общими размерами заключается в обеспечении одинаковых пропорций различным элементам управления Grid.

Для демонстрации групп с общими размерами рассмотрим пример, показанный на рис. 16. Это окно оснащено двумя объектами Grid – один в верхней части окна (с тремя колонками) и один в его нижней части (с двумя колонками). Размер левой крайней колонки первого Grid устанавливается пропорционально ее содержимому (длинной

текстовой строке). Левая крайняя колонка второго Grid имеет в точности ту же ширину, хотя имеет меньшее содержимое. Дело в том, что они входят в одну размерную группу. Независимо от того, какое содержимое будет помещено в первую колонку первого Grid, первая колонка второго Grid останется синхронизированной.

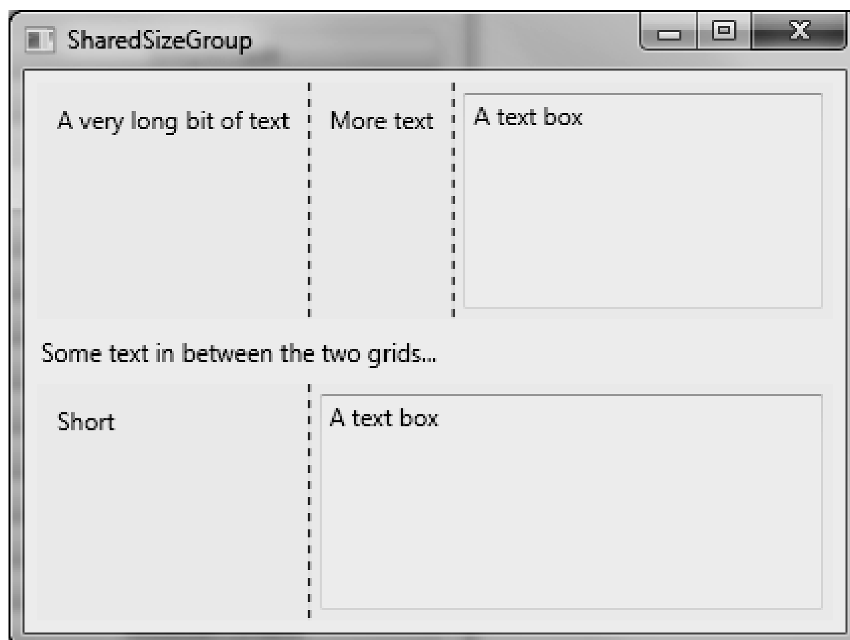

Рис. 15. Группы с общими размерами

Как демонстрирует этот пример, колонки с общими размерами могут принадлежать к разным Grid. В этом примере верхний Grid имеет на одну колонку больше, и потому оставшееся пространство в нем распределяется иначе. Аналогично колонки с общими размерами могут занимать разные позиции, так что имеется возможность создать отношение между первой колонкой одного Grid и второй колонкой другого. И, очевидно, колонки при этом могут иметь совершенно разное содержимое.

Использование группы с общими размерами аналогично созданию одного определения колонки (или строки), используемого в более чем одном месте. Это не просто однонаправленная копия одной колонки в другую. В этом можно убедиться, изменив содержимое разделенной колонки второго Grid. Колонка в первом Grid будет автоматически удлинена для сохранения соответствия.

Можно даже добавить GridSplitter к одному из объектов Grid. Когда пользователь будет изменять размер колонки в одном Grid, то соответствующая разделенная колонка из второго Grid также будет синхронно менять свой размер.

Создать группы с общими размерами просто. Достаточно лишь установить свойство SharedSizeGroup в обеих колонках, используя строку соответствия. В текущем примере обе колонки используют группу по имени TextLabel.

```
<Grid Grid.IsSharedSizeScope="True" Margin="3">
<Grid.RowDefinitions>
<RowDefinition></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Grid Grid.Row="0" Margin="3" Background="LightYellow" Show-
GridLines="True">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto" SharedSize-
Group="TextLabel"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition></ColumnDefinition>
</Grid.ColumnDefinitions>
<Label Margin="5">A very long bit of text</Label>
<Label Grid.Column="1" Margin="5">More text</Label>
<TextBox Grid.Column="2" Margin="5">A text box</TextBox>
</Grid>
<Label Grid.Row="1" >Some text in between the two gr-
ids...</Label>
<Grid Grid.Row="2" Margin="3" Background="LightYellow"  Show-
GridLines="True">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto" SharedSize-
Group="TextLabel"></ColumnDefinition>
```

```
<ColumnDefinition></ColumnDefinition>
</Grid.ColumnDefinitions>
<Label Margin="5">Short</Label>
<TextBox Grid.Column="1" Margin="5">A text box</TextBox>
</Grid>
</Grid>
```

Отметим, что группы с общими размерами не являются глобальными для всего приложения, потому что более одного окна могут случайно использовать одно и то же имя. Чтобы разделить группу, необходимо явно установить прикрепленное свойство Grid.IsSharedSizeScope в true в контейнере высшего уровня, содержащем объекты Grid с колонками с общими размерами. В текущем примере верхний и нижний Grid входят в другой Grid, предназначенный для этой цели, хотя можно использовать другой контейнер, такой как DockPanel или StackPanel.

3.6. UniformGrid

Существует элемент, аналогичный сетке, но нарушающий правила, рассмотренные выше – это UniformGrid. В отличие от Grid, элемент UniformGrid не требует и не поддерживает предопределенных колонок и строк. Вместо этого разработчик просто устанавливает свойства Rows и Columns для установки его размеров. Каждая ячейка всегда имеет одинаковый размер, потому что доступное пространство делится поровну. И, наконец, элементы помещаются в соответствующую ячейку на основе порядка их определения. Нет никаких прикрепленных свойств Row и Column, и никаких пустых ячеек.

Приведем пример, наполняющий UniformGrid четырьмя кнопками:

```
<UniformGrid Rows="2" Columns="2">
<Button>Top Left</Button>
<Button>Top Right</Button>
<Button>Bottom Left</Button>
```

```
<Button>Bottom Right</Button>
</UniformGrid>
```

UniformGrid используется намного реже, чем Grid. Элемент Grid – это инструмент общего назначения для создания компоновки окон, от самых простых до самых сложных. UniformGrid намного более специализированный контейнер компоновки, который в первую очередь предназначен для размещения элементов в жесткой сетке (например, для построения игрового поля для игр).

3.7. Координатная компоновка с помощью Canvas

Единственный контейнер компоновки, который еще не был рассмотрен – это Canvas. Он позволяет размещать элементы, используя точные координаты, что является плохим выбором при проектировании богатых управляемых данными форм и стандартных диалоговых окон. Однако этот контейнер является ценным инструментом, если нужно построить нечто другое – например, поверхность рисования для инструмента построения диаграмм. Canvas также является наиболее легковесным из контейнеров компоновки. Это объясняется тем, что он не включает в себя никакой сложной логики компоновки, согласовывающей размерные предпочтения своих дочерних элементов. Вместо этого он просто располагает их в указанных разработчиком позициях с точными размерами, которые нужны разработчику.

Чтобы позиционировать элемент на Canvas, следует устанавливать прикрепленные свойства Canvas.Left и Canvas.Top. Свойство Canvas.Left задает количество единиц измерения между левой гранью элемента и левой границей Canvas. Свойство Canvas.Top устанавливает количество единиц измерения между вершиной элемента и левой границей Canvas.

Также можно устанавливать размер элемента явно, используя его свойства Width и Height. Это чаще применяется при использовании Canvas, чем с другими панелями, потому что Canvas не имеет

собственной логики компоновки. К тому же Canvas часто применяется, когда необходим точный контроль расположения комбинации элементов. Если не устанавливать свойства Width и Height, элемент получит желательный для него размер, т.е. он станет достаточно большим, чтобы вместить свое содержимое.

Ниже приведен пример простого Canvas, включающего четыре кнопки.

```
<Canvas>
<Button Canvas.Left="10" Canvas.Top="10">(10,10)</Button>
<Button Canvas.Left="120" Canvas.Top="30">(120,30)</Button>
<Button Canvas.Left="60" Canvas.Top="80" Width="50"
Height="50">(60,80)</Button>
<Button Canvas.Left="70" Canvas.Top="120" Width="100"
Height="50">(70,120)</Button>
</Canvas>
```

На рис. 17 показан результат.

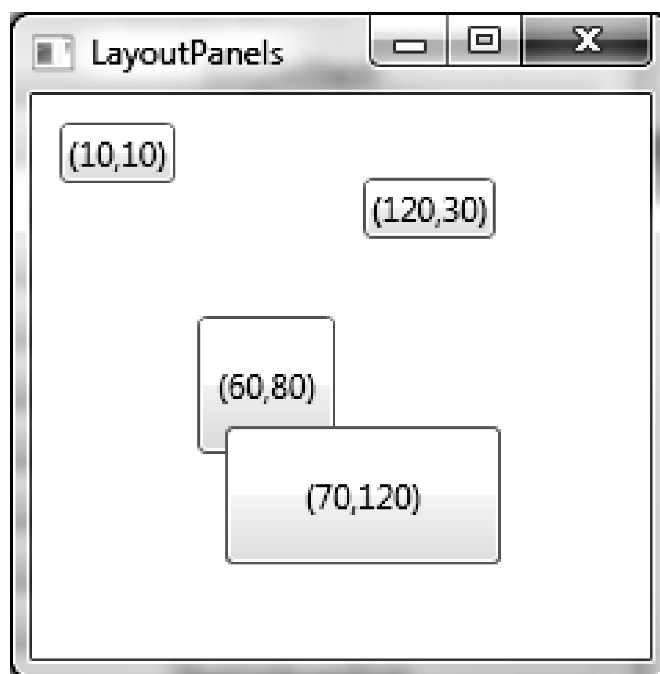

Рис. 16. Явное позиционирование элементов в Canvas

Если изменить размеры окна, то Canvas растянется для заполнения всего доступного пространства, но ни один из элементов управления на его поверхности не изменит своего положения и размера.

Canvas не включает никаких средств привязки или стыковки, которые имеются в координатных компоновках Windows Forms. Отчасти это объясняется легковесностью Canvas. Другая причина заключается в том, что эта особенность позволяет предотвратить использование Canvas для тех целей, для которых он не предназначен.

Подобно любому другому контейнеру компоновки, Canvas может вкладываться внутрь пользовательского интерфейса. Это значит, что можно применять Canvas для рисования более детализированного содержимого в части окна, используя стандартные панели WPF для остальной части элементов.

Если имеется более одного перекрывающегося элемента, можно установить прикрепленное свойство Canvas.ZIndex для управления их расположением.

Обычно все добавляемые элементы имеют одинаковый ZIndex, равный 0. Когда элементы имеют одинаковый ZIndex, они отображаются в том порядке, в каком они представлены в коллекции Canvas.Children, который основан на порядке их определения в разметке XAML. Элементы, объявленные позже, в разметке отображаются поверх элементов, объявленных ранее.

Можно передвинуть любой элемент на более высокий уровень, увеличив его ZIndex. Это объясняется тем, что элементы с большим ZIndex всегда появляются поверх элементов с меньшим ZIndex. Используя эту технику, можно обратить часть компоновки из предыдущего примера:

```
<Button Canvas.Left="60" Canvas.Top="80" Width="50" Height="50"
Canvas.ZIndex="1">(60,80)</Button>
<Button Canvas.Left="70" Canvas.Top="120" Width="100"
Height="50">(70,120)</Button>
```

Отметим, чтосами значения, используемые для свойства Canvas.ZIndex, не важны, определяющим является отношение значений ZIndex разных элементов между собой.

3.8. InkCanvas

WPF также включает элемент InkCanvas, который подобен Canvas в одних отношениях и совершенно отличается в других. Подобно Canvas, элемент InkCanvas определяет четыре прикрепленных свойства, которые можно применить к дочерним элементам для координатного позиционирования (Top, Left, Bottom и Right). Однако лежащий в его основе механизм существенно отличается. Фактически, InkCanvas не наследуется от Canvas и даже не наследуется от базового класса Panel. Вместо этого он наследуется непосредственно от FrameworkElement.

Главное предназначение InkCanvas заключается в обеспечении перьевого ввода, используемого в смартфонах, планшетах и наладонных ПК. Однако InkCanvas работает с мышью точно так же, как и с пером. Поэтому пользователь может рисовать линии или выбирать и манипулировать элементами в InkCanvas с применением мыши.

InkCanvas в действительности содержит две коллекции дочернего содержимого. Рассмотренная выше коллекция Children содержит произвольные элементы – как и Canvas. Каждый элемент может быть позиционирован на основе свойств Top, Left, Bottom и Right. Коллекция Strokes содержит объекты System.Windows.Ink.Stroke, представляющие графический ввод, который рисует пользователь в InkCanvas. Каждая линия или кривая, которую рисует пользователь, становится отдельным объектом Stroke. Благодаря этим двум коллекциям, можно использовать InkCanvas для того, чтобы позволить пользователю аннотировать содержимое (хранящееся в коллекции Children) пометками (хранящимися в коллекции Strokes).

Например, на рис. 18 показан элемент InkCanvas, содержащий картинку, аннотированную дополнительными пометками. Вот разметка InkCanvas из этого примера, которая определяет изображение:

```
<InkCanvasName="inkCanvas" Grid.Row="1"
Background="LightYellow" Editing-
```

```
Mode="{BindingElementName=lstEditingMode, Path=SelectedItem}">
<Button InkCanvas.Top="10" InkCanvas.Left="10">Hello</Button>
<Image Source="cat.jpg" InkCanvas.Top="10" InkCan-
vas.Left="10"Width="287" Height="319"></Image>
</InkCanvas>
```

Пометки нарисованы пользователем во время выполнения.

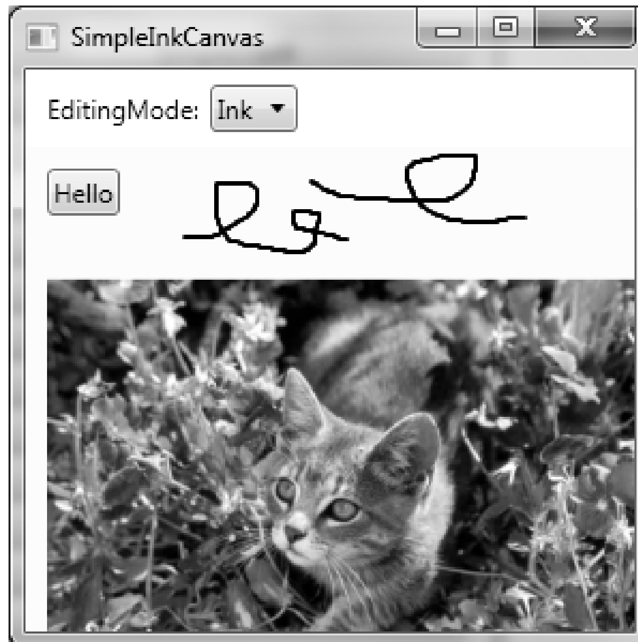

Рис. 17. InkCanvasс изображением и пометками

InkCanvas может применяться несколькими существенно отличающимися способами, в зависимости от значения, установленного для свойства InkCanvas.EditingMode. Возможные варианты этого значения перечислены в таблице 5.

Таблица 5

Значения перечисления InkCanvas.EditingMode

Имя	Описание
Ink	InkCanvas позволяет пользователю рисовать аннотации. Это режим по умолчанию. Когда пользователь рисует мышью или пером, появляются штрихи.
GestureOnly	InkCanvas не позволяет пользователю рисовать аннотации, обрабатывает некоторыепредопределенные жесты (gestures), такие как перемещение пера в одном направлении или подчеркивание содержимого. Полный список жестов определен в перечислении System.Windows.Ink.ApplicationGesture.
InkAndGesture	InkCanvas позволяет пользователю рисовать штриховые аннотации и также распознает предопределенные жесты.
EraseByStroke	InkCanvas удаляет штрих при щелчке.
EraseByPoint	InkCanvas удаляет часть точку штриха при щелчке по соответствующей его части.
Select	InkCanvas позволяет пользователю выбирать элементы, хранящиеся в коллекции children. Чтобы выбрать элемент, пользователь должен щелкнуть на нем или обвести «лассо» выбора вокруг него. Как только элемент выбран, его можно перемещать, изменять размер или удалять.
None	InkCanvas игнорирует ввод с помощью мыши или пера.

InkCanvas инициирует события при изменении режима редактирования (ActiveEditingModeChanged), обнаружении жеста в режимах GestureOnly или InkAndGesture (Gesture), рисовании штриха (Stroke-Collected), стирании штриха (StrokeErasing и StrokeErased), а также при выборе элемента или изменении его в режиме Select (Selection-Changing, SelectionChanged, SelectionMoving, SelectionMoved, SelectionResizing и SelectionResized). События, оканчивающиеся на -ing, представляют действие, которое начинается, но может быть отменено установкой свойства Cancel объекта EventArgs.

В режиме Select элемент InkCanvas предоставляет довольно

удобную поверхность проектирования для перетаскивания содержимого и различных манипуляций им. На рис. 19 показан элемент управления Button в InkCanvas, когда он был выбран (слева) и затем перемещен и увеличен (справа).

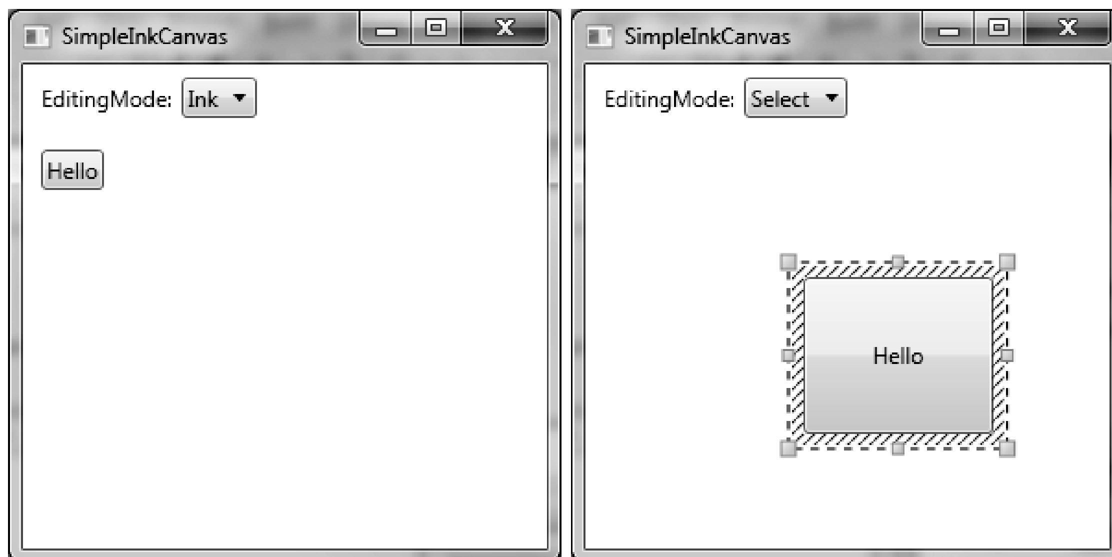

Рис. 18. Изменение размера элемента в InkCanvas

3.9. Примеры компоновки

В данном разделе будут рассмотрены несколько завершенных примеров компоновки.

3.9.1. Колонка настроек

Контейнеры компоновки, подобные Grid, значительно упрощают задачу создания общей структуры окна. Например, рассмотрим окно с настройками, показанное на рис. 20. Это окно располагает свои индивидуальные компоненты – метки, текстовые поля и кнопки – в табличной структуре.

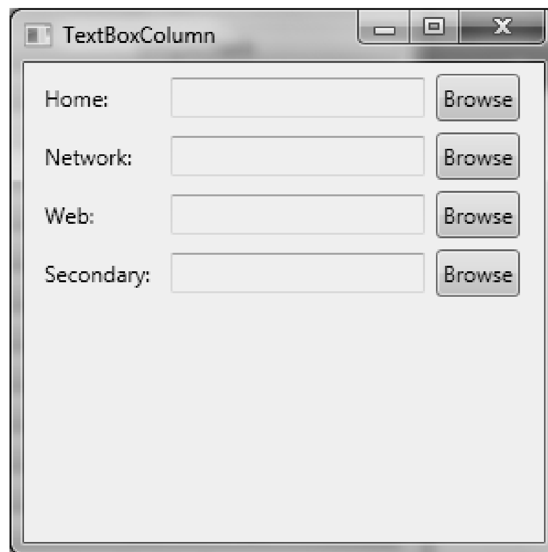

Рис. 19. Настройка папки в колонке

Чтобы создать эту таблицу, следует начать с определения строк и колонок сетки. Строки достаточно просты – размер каждой просто определяется по высоте содержимого. Поэтому вся строка получит высоту самого большого элемента, которым в данном случае является кнопка Browse (Обзор) из третьей колонки.

Далее нужно создать колонки. Размер первой и последней колонки определяется так, чтобы вместить их содержимое (текст метки и кнопку Browse соответственно). Средняя колонка получает все оставшееся пространство, а это значит, что она будет расти при увеличении размера окна, предоставляя больше места, чтобы видеть выбранную папку.

Имея базовую структуру, нужно просто разместить элементы по правильным ячейкам. Однако также следует тщательно продумать поля и выравнивание. Каждый элемент нуждается в базовом поле (подходящим значением для него будет 3 единицы), чтобы создать небольшой отступ от края окна. Вдобавок метка и текстовое поле должны быть центрированы по вертикали, потому что их высота меньше, чем у кнопки Browse. И, наконец, текстовое поле должно использовать режим автоматической установки размера, растягиваясь для того, чтобы вместить всю колонку.

Отметим один факт, который связан с тем, насколько гибким является это окно благодаря использованию элемента управления Grid. Ни один из индивидуальных элементов – ни метки, ни текстовые поля, ни кнопки – не имеют жестко закодированных позиций и размеров. В результате можно легко вносить изменения в сетку, просто изменяя элементы ColumnDefinition. Более того, если добавить строку, которая имеет более длинный текст метки, вся сетка будет откорректирована автоматически, сохраняя согласованность, включая строки, которые были добавлены ранее. И если возникнет необходимость добавить элементы между существующими строками, такие как разделительные линии, чтобы отделить друг от друга разные разделы окна, можно сохранить те же колонки, но использовать свойство ColumnSpan для растяжения единственного элемента на большую область.

Полный код разметки данного окна приведен ниже:

```
<Grid Margin="3,3,10,3">
<Grid.RowDefinitions>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
     <ColumnDefinition Width="Auto"></ColumnDefinition>
     <ColumnDefinition Width="*" MinWidth="50" Max-
Width="800"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
<Label Grid.Row="0" Grid.Column="0" Margin="3"
         VerticalAlignment="Center">Home:</Label>
<TextBox Grid.Row="0" Grid.Column="1" Margin="3"
             Height="Auto"  VerticalAlign-
ment="Center"></TextBox>
```

```
<Button Grid.Row="0" Grid.Column="2" Margin="3" Pad-
ding="2">Browse</Button>
<Label Grid.Row="1" Grid.Column="0" Margin="3"
             VerticalAlignment="Center">Network:</Label>
<TextBox Grid.Row="1" Grid.Column="1" Margin="3"
             Height="Auto"  VerticalAlign-
ment="Center"></TextBox>
<Button Grid.Row="1" Grid.Column="2" Margin="3" Pad-
ding="2">Browse</Button>
<Label Grid.Row="2" Grid.Column="0" Margin="3"
             VerticalAlignment="Center">Web:</Label>
<TextBox Grid.Row="2" Grid.Column="1" Margin="3"
             Height="Auto"  VerticalAlign-
ment="Center"></TextBox>
<Button Grid.Row="2" Grid.Column="2" Margin="3" Pad-
ding="2">Browse</Button>
<Label Grid.Row="3" Grid.Column="0" Margin="3"
             VerticalAlignment="Center">Secondary:</Label>
<TextBox Grid.Row="3" Grid.Column="1" Margin="3"
             Height="Auto"  VerticalAlign-
ment="Center"></TextBox>
<Button Grid.Row="3" Grid.Column="2" Margin="3" Pad-
ding="2">Browse</Button>
</Grid>
```

3.9.2. Динамическое содержимое

Как демонстрирует показанная колонка настроек, окна, использующие контейнеры компоновки WPF, легко поддаются изменениям и адаптации по мере развития приложения. И преимущество этой гибкости проявляется не только во время проектирования. Это также ценное приобретение, если есть необходимость отображения динамически изменяющегося содержимого.

Примером может служить локализованный текст – текст, который появляется в пользовательском интерфейсе и нуждается в пере-

воде на разные языки для разных регионов. В приложениях, опирающихся на координатные системы, изменение текста может разрушить внешний вид окна – в частности, потому, что краткие предложения английского языка становятся существенно длиннее на многих других языках. Даже если элементам позволено изменять свои размеры, чтобы вместить больший текст, это может нарушить общий баланс окна.

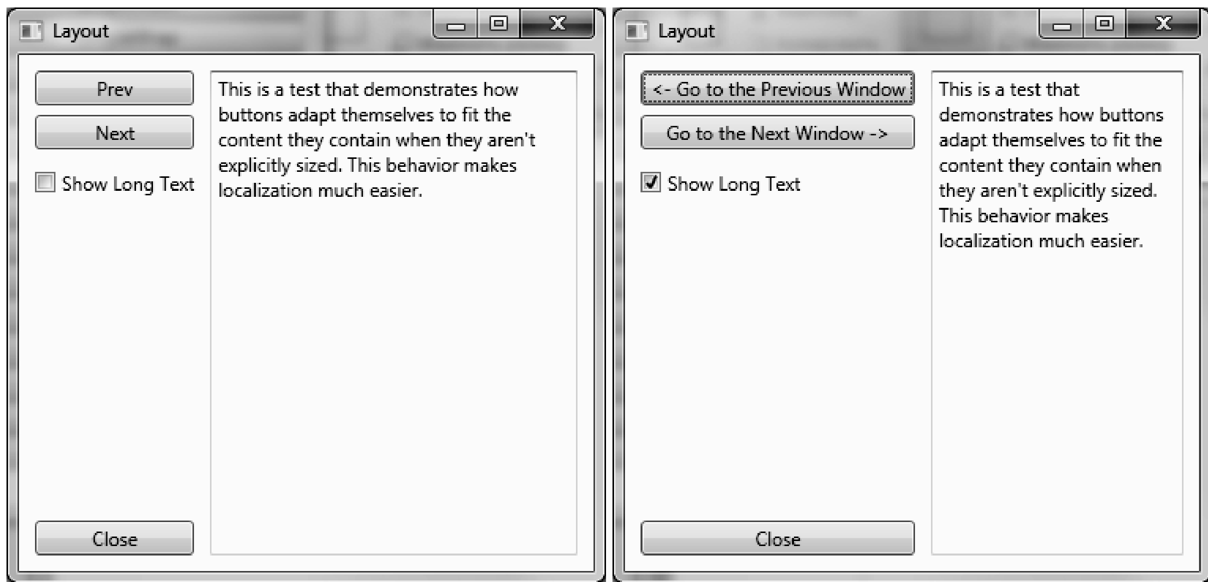

Рис. 20. Самонастраивающееся окно

На рис. 21 показано, как можно избежать этих неприятностей, если разумно использовать контейнеры компоновки WPF. В этом примере пользовательский интерфейс имеет опции краткого и длинного текста. Когда используется длинный текст, кнопки, содержащие текст, изменяют свой размер автоматически, расталкивая соседнее содержимое. И поскольку кнопки измененного размера разделяют один и тот же контейнер компоновки (в данном случае – колонку таблицы), весь раздел пользовательского интерфейса изменяет свой размер. В результате получается, что кнопки сохраняют согласованный размер – размер самой большой из них.

Полная разметка данного окна выглядит следующим образом:

```
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="*"></RowDefinition>
```

```
<RowDefinition Height="Auto"></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition Width="*"></ColumnDefinition>
</Grid.ColumnDefinitions>
<StackPanel Grid.Row="0" Grid.Column="0">
<Button Name="cmdPrev" Margin="10,10,10,3">Prev</Button>
<Button Name="cmdNext" Margin="10,3,10,3">Next</Button>
<CheckBox Name="chkLongText" Margin="10,10,10,10"
Checked="chkLongText_Checked" Un-
checked="chkLongText_Unchecked">Show Long Text</CheckBox>
</StackPanel>
<TextBox Grid.Row="0" Grid.Column="1" Margin="0,10,10,10"
TextWrapping="WrapWithOverflow" Grid.RowSpan="2">This is a test
that demonstrates
how buttons adapt themselves to fit the content they contain
when they aren't
explicitly sized. This behavior makes localization much easi-
er.</TextBox>
<Button Grid.Row="1" Grid.Column="0" Name="cmdClose" Mar-
gin="10,3,10,10">Close</Button>
</Grid>
```

3.9.3. Модульный пользовательский интерфейс

Многие из контейнеров компоновки успешно помещают содержимое в доступное пространство – так поступают StackPanel, DockPanel и WrapPanel. Одно из преимуществ этого подхода заключается в том, что он позволяет строить действительно модульные интерфейсы. Другими словами, можно подключать разные панели с соответствующими секциями пользовательского интерфейса, которые следует показать, и пропускать те, которые в данный момент не нужны. Приложение в целом может подстраивать себя соответствующим образом – подобно портальному сайту в Web.

Подобный пользовательский интерфейс продемонстрирован на рис. 22. Здесь в WrapPanel помещается несколько отдельных панелей. Пользователь может выбрать те панели, которые должны быть видимыми, используя флажки в верхней части окна.

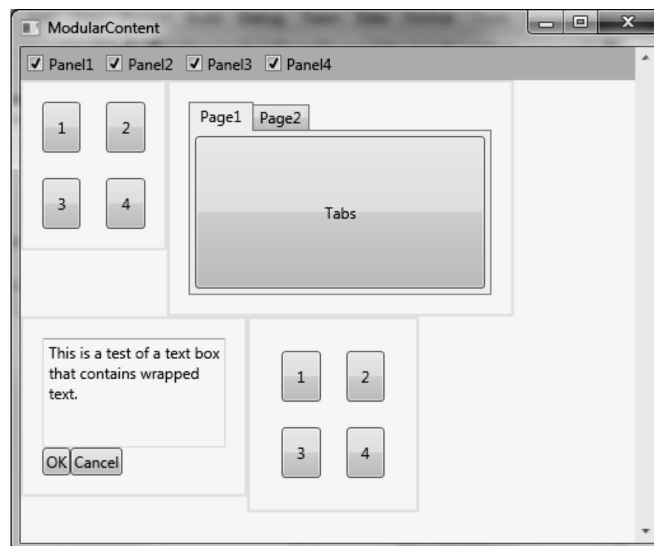

Рис. 21. Серии панелей в WarpPanel

Если другие панели скрыты, оставшиеся реорганизуют себя, заполняя доступное пространство (и порядок, в котором они объявлены). На рис. 23 показана другая организация тех же панелей.

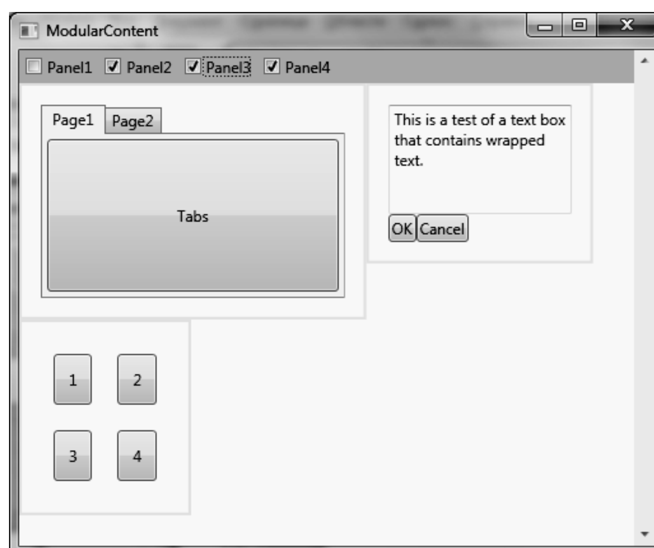

Рис. 22. Скрытие панелей в WarpPanel

Чтобы скрыть или показать индивидуальные панели, нужен небольшой фрагмент кода, обрабатывающего щелчки на флажках. Код состоит в установке свойства Visibility:

```
panel.Visibility=Visibility.Collapsed;
```

Свойство Visibility – это часть базового класса UIElement и потому поддерживается почти всеми объектами, которые помещаются в окно WPF. Оно принимает одно из трех значений перечисления System.Windows.Visibility:

- Visible: элемент появляется в окне в нормальном виде;
- Collapsed:элемент не отображается и не занимает места;
- Hidden: Элемент не отображается, но место за ним резервируется.

4. СОДЕРЖИМОЕ

Выше была рассмотрена система компоновки в WPF, которая позволяет компоновать окно, помещая компоненты в специализированные контейнеры компоновки. Благодаря этой системе даже простое окно разбивается на вложенные серии контейнеров Grid, StackPanel и DockPanel. Пройдя всю серию вложений, в конечном итоге будут обнаружены видимые элементы внутри различных контейнеров.

Контейнеры компоновки не являются единственным примером вложенных элементов. WPF построена на новой модели содержимого, позволяющей помещать компоненты в другие элементы, которые в любом другом случае используются как обычные элементы. Благодаря этой технологии можно брать многие простые элементы управления, такие как кнопки, и помещать в них картинки, векторные формы и даже контейнеры компоновки. Эта модель содержимого является одной из особенностей WPF, которые придают ей высокую степень гибкости.

4.1. Элементы управления содержимым

В разделе 2 была показана иерархия классов, которая образует основу WPF. Также были описаны различия, существующие между

собственно элементами (к которым относится все, что помещается в окно WPF) и элементами управления (к которым относятся специализированные элементы, являющиеся наследниками класса System.Windows.Controls.Control). В WPF элемент управления обычно описывается как элемент, который может получать фокус ввода и принимать данные, вводимые пользователем – в качестве примера можно привести текстовое поле или кнопку. Однако отличие иногда бывает очень расплывчатым. ToolTip считается элементом управления, поскольку он появляется и исчезает в зависимости от перемещений указателя мыши. Label считается элементом управления, поскольку он поддерживает мнемонические команды – клавиши быстрого доступа, передающие фокус связанным элементам управления.

Элементы управления содержимым (content control) – это специализированный тип элементов управления, которые могут хранить и отображать какую-то порцию содержимого. С технической точки зрения элемент управления содержимым является элементом управления, который может включать один вложенный элемент. Этим он отличается от контейнера компоновки, который может хранить сколь угодно много вложенных элементов. Очевидно, что можно поместить большой объем содержимого в один элемент управления содержимым – для этого потребуется упаковать все содержимое в один контейнер, такой как StackPanel или Grid. Например, класс Window сам является элементом управления содержимым. Очевидно, что окна часто хранят большие объемы содержимого, которое, однако, помещается в один контейнер верхнего уровня.

Как было показано выше, все контейнеры компоновки WPF являются наследниками класса Panel, что позволяет им хранить множество элементов. Точно так же, все элементы управления содержимым являются наследниками абстрактного класса ContentControl. Иерархия классов показана на рис. 24.

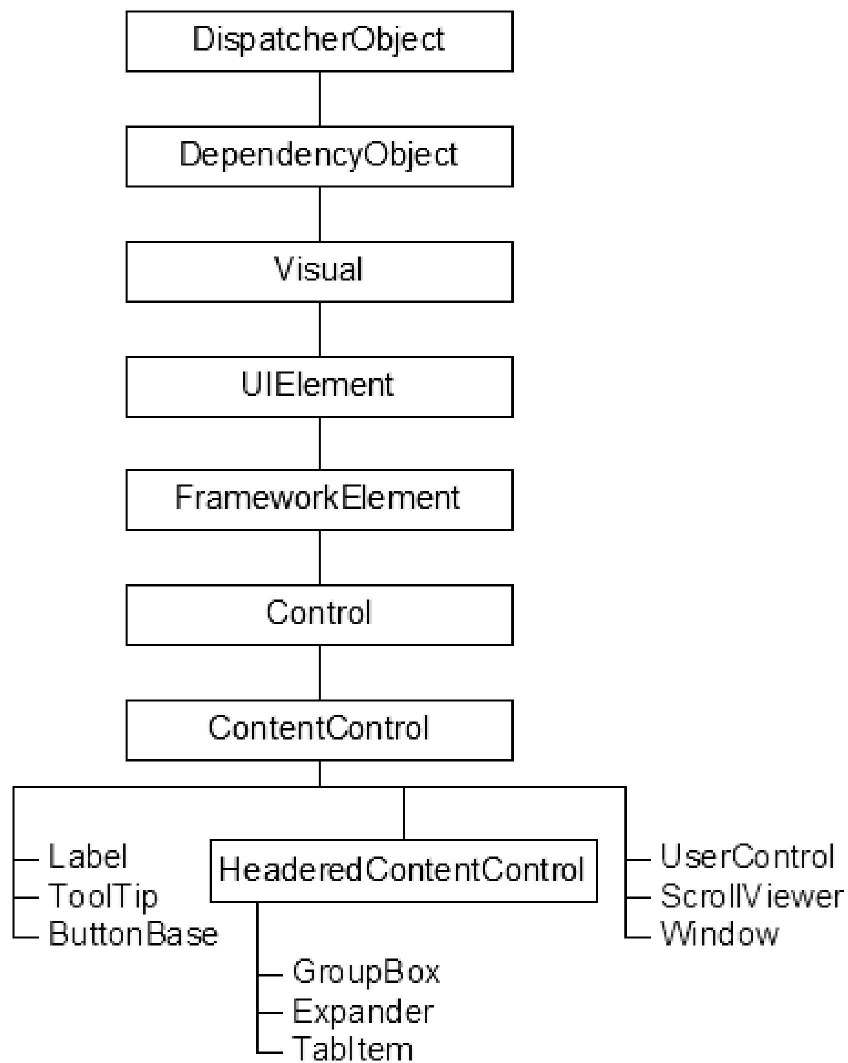

Рис. 23. Иерархия элементов управления содержимым

Как можно видеть на рис. 24, некоторые элементы управления на самом деле являются элементами управления содержимым, в том числе Label и ToolTip. Кроме того, все типы кнопок являются элементами управления содержимым, включая Button, RadioButton и CheckBox. Существует еще несколько специализированных элементов управления содержимым, такие как классы ScrollViewer (позволяет создавать прокручивающиеся панели) и UserControl (позволяет повторно использовать специальное группирование элементов управления). Класс Window, который служит для представления каждого окна в приложении, сам по себе является элементом управления содержимым.

Кроме того, существует еще ряд элементов управления содержимым, которые являются наследниками класса HeaderedContentControl. Эти элементы управления имеют область содержимого и область заголовка, которые могут применяться для отображения некоторой разновидности заголовка. К этим элементам управления относятся GroupBox, TabItem (страница в TabControl) и Expander.

4.2. Свойство Content

В то время как класс Panel добавляет коллекцию Children для хранения вложенных элементов, класс ContentControl добавляет свойство Content, которое принимает один объект. Свойство Content поддерживает любой тип объектов, хотя все объекты оно разделяет на две группы, каждая из которых обрабатывается по-разному:

- Объекты, которые не являются наследниками класса UIElement. Элемент управления содержимым вызывает метод ToString() для получения текста для этих элементов управления, после чего отображает этот текст;
- Объекты, которые являются наследниками класса UIElement. Эти объекты (к ним относятся все визуальные элементы, которые являются частью WPF) отображаются внутри элемента управления содержимым с помощью метода UIElement.OnRender().

Отметим, что с технической точки зрения метод OnRender() не рисует объект – он просто генерирует графическое представление, которое WPF отображает на экране по мере необходимости.

Для демонстрации работы этого механизма, рассмотрим простую кнопку. В примерах с кнопками, которые были продемонстрированы выше, была такая строка:

```
<Button Margin="3">Text button</Button>
```

Эта строка задается как содержимое кнопки и отображается на поверхности кнопки. Однако задачу можно усложнить, поместив в

кнопку другие элементы. Например, с помощью класса image в нее можно поместить изображение:

```
<Button Margin="3">
<Image Source="happyface.jpg" Stretch="None" />
</Button>
```

Как вариант, можно комбинировать текст и изображения, поместив их в контейнер компоновки, например:

```
<Button Margin="3">
<StackPanel>
<TextBlock Margin="3">Image and text button</TextBlock>
<Image Source="happyface.jpg" Stretch="None" />
<TextBlock Margin="3" >Courtesy of the StackPanel</TextBlock>
</StackPanel>
</Button>
```

Отметим, что в этом примере вместо элемента управлении Label используется элемент управления TextBlock. TextBlock представляет собой облегченный текстовый элемент, поддерживающий компоновку текста, но не поддерживающий использование клавиш быстрого доступа.Также, в отличие от метки Label, TextBlock не является элементом управления содержимым.

Если есть необходимость создать действительно экзотическую кнопку, можно поместить в нее другие элементы управления содержимым, такие как текстовые поля и кнопки (а в них можно разместить другие элементы). Пример такой кнопки:

```
<Button Padding="3" Margin="3" HorizontalContentAlign-
ment="Stretch">
<StackPanel>
<TextBlock Margin="3">Type something here:</TextBlock>
<TextBox Margin="3" HorizontalAlignment="Stretch">Text box in a
button</TextBox>
</StackPanel>
</Button>
```

Смысла в подобном варианте интерфейса маловато, но сама возможность построения такого интерфейса существует. На рис. 25 показаны рассмотренные выше кнопки:

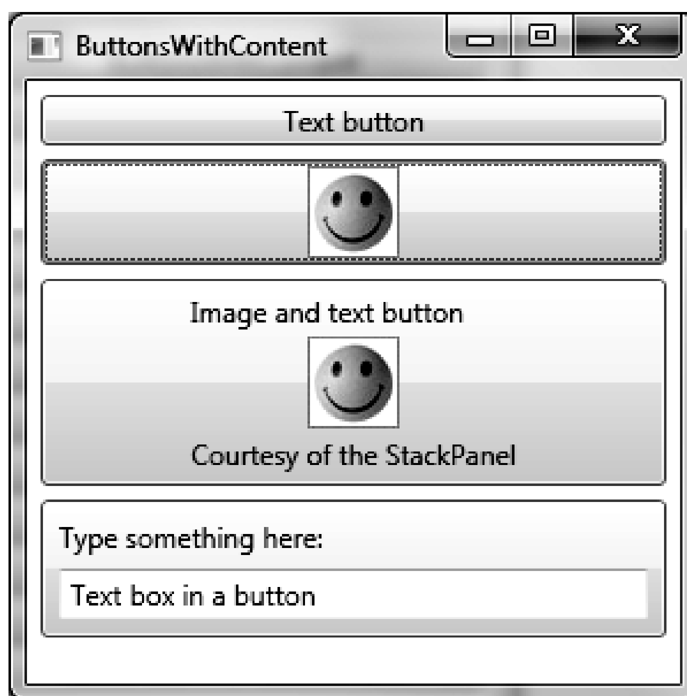

Рис. 24. Кнопки с разнотипным вложенным содержимым

Это та же модель содержимого, которая рассматриваласьвыше. Как и Button, класс Window допускает использование вложенных элементов, в качестве которых могут выступать порции текста, произвольные объекты или элемент.

Помимо свойства Content класс ContentControl определяет еще некоторые элементы. Он включает свойство HasContent, которое возвращает значение true, если в элементе управления имеется содержимое, и свойство ContentTemplate, которое позволяет создать шаблон, сообщающий элементу управления о том, как нужно отображать объект, который в любом другом случае не будет распознан. С помощью ContentTemplate можно более интеллектуальным способом отображать объекты, не являющиеся наследниками класса UIElement. Вместо того чтобы просто вызывать метод ToString() для получения строки, можно задать разные значения свойства, чтобы упорядочить их в более сложной разметке.

4.3. Выравнивание содержимого

В разделе 3.3.2. Выравнивание» речь шла о том, как осуществляется выравнивание разных элементов управления в контейнере с помощью свойств HorizontalAlignment и VerticalAlignment, которые определены в базовом классе FrameworkElement. Однако после того как элемент управления получает содержимое, сразу возникает вопрос об его организации. Разработчику нужно решить, как будет выравниваться содержимое внутри элемента управления. Для этой цели используются свойства HorizontalContentAlignment и VerticalContentAlignment.

Свойства HorizontalContentAlignment и VerticalContentAlignment поддерживают те же значения, что и свойства HorizontalAlignment и VerticalAlignment. Это означает, что можно выровнять содержимое вдоль какого-нибудь края (Top, Bottom, Left или Right) или по центру (Center), либо можете растянуть его так, чтобы заполнить все доступное пространство (Stretch). Эти настройки применяются непосредственно к вложенному элементу содержимого, хотя есть возможность задать множество уровней вложения, получив более изощренную компоновку.

В разделе 3.3.3. Поля» было описано свойство Margin, которое позволяет добавлять пустое пространство между соседними элементами. Элементы управления содержимым используют дополнительное свойство Padding, которое вставляет пустое пространство между краями элемента управления и краями содержимого. Чтобы посмотреть разницу, сравним следующие две кнопки:

```
<Button>AbsolutelyNoPadding</Button>
<Button Padding="3">Well Padded</Button>
```

В кнопке, в которой нет заполнения, текст начинается от самого края кнопки. В кнопке, в которой с каждого края заполнены три единицы пространства, текст выглядит более привлекательно. Эту разницу можно увидеть на рис. 26.

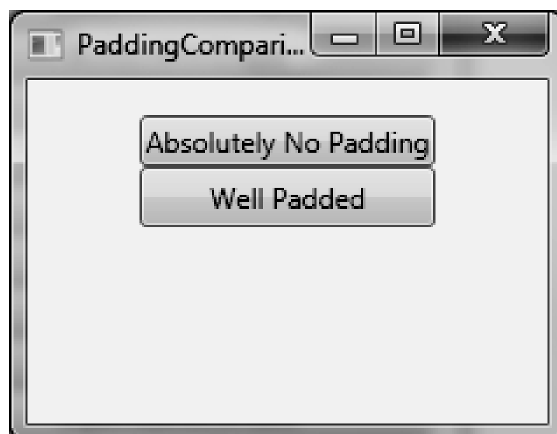

Рис. 25.Заполнение содержимого кнопки

4.4. Модель содержимого в WPF

Может возникнуть сомнение относительно того, действительно ли модель содержимого, используемая в WPF, обладает серьезными преимуществами. В конце концов, можно поместить изображение внутрь кнопки, а внедрять другие элементы управления и целые панели компоновки вряд ли имеет смысл. И все-таки в пользу этой модели имеются подходящие доводы.

Рассмотрим пример, показанный на рис. 25, на котором элемент Image находится внутри элемента Button. Этот подход не является идеальным, поскольку битовые образы очень сильно зависят от разрешения. На экране монитора с высоким разрешением битовый образ может выглядеть размытым, так как WPF добавляет большое количество пикселей при интерполяции с целью сохранения корректных размеров. В более изощренных интерфейсах WPF вместо битовых образов применяется комбинация векторных форм.

Этот подход хорошо уживается вместе с моделью элементов управления содержимым. Поскольку класс Button является элементом управления содержимым, есть возможность помещать в него не только фиксированные битовые образы, но и включать в него содержимое другого типа. Например, с помощью классов из пространства имен System.Windows.Shapes можно нарисовать векторное изображение внутри кнопки. Ниже показан пример, в котором создается кнопка с двумя ромбообразными формами (см. рис. 27):

```
<Button Margin="10">
<Grid>
<Polygon Points="100,25 125,0 200,25 125,50"
Fill="Yellow" />
<Polygon Points="100,25 75,0 0,25 75,50"Fill="Purple"/>
</Grid>
</Button>
```

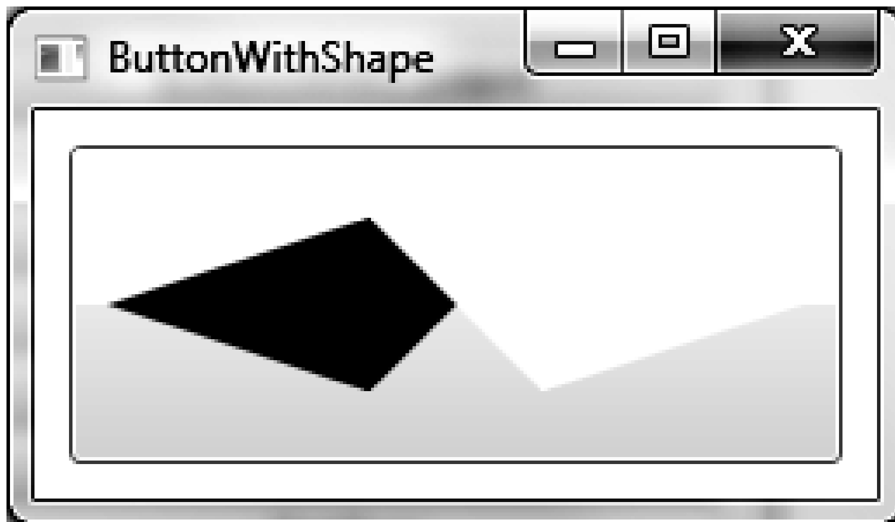

Рис. 26. Кнопка с разнообразными формами

Очевидно, что в данном случае гораздо проще использовать модель вложенного содержимого, чем добавлять дополнительные свойства в класс Button для поддержки различных типов содержимого. Модель вложенного содержимого не просто более гибкая – она позволяет упростить интерфейс класса Button. А поскольку все элементы управления содержимым поддерживают вложение содержимого одинаковым образом, то отпадает необходимость добавлять различные свойства содержимого во многие классы.

Фактически, модель вложенного содержимого является неким компромиссом. Она упрощает модель классов для элементов, поскольку в ее случае не нужно использовать дополнительные уровни наследования, чтобы добавить свойства для различного типа содержимого. Тем не менее, необходимо работать с чуть более сложной моделью объектов – элементами, которые могут быть построены из других вложенных элементов.

Отметим, что не всегда можно получить желаемый результат, заменяя содержимое элемента управления. Например, даже если поместить любое содержимое в кнопку, некоторые детали все равно никогда не изменятся, такие как затененный фон кнопки, скругленные границы и эффект при наведении указателя мыши, при котором поверхность кнопки осветляется, когда над ней находится указатель. Тем не менее, изменить встроенные детали можно путем применения нового шаблона элементов управления.

4.5. Специализированные контейнеры

В данном разделе кратко рассмотрим некоторые элементы управления содержимым: ScrollViewer, GroupBox, TabItem и Expander. Каждый из этих элементов управления создан для того, чтобы можно было придать форму крупным порциям пользовательского интерфейса. Однако в связи с тем, что эти элементы управления могут содержать только один элемент, они применяются в сочетании с контейнером компоновки.

4.5.1. Элемент управления ScrollViewer

Вышерассматривались некоторые контейнеры компоновки. Однако ни один из них не обеспечивает поддержкупрокрутки, которая будет являться ключевой возможностью, если необходимо помещать большие объемы содержимого в ограниченный объем пространства. В WPF поддержка прокрутки обеспечивается специальнымкомпонентом – элементом управления содержимым ScrollViewer.

Чтобы обеспечить поддержку прокрутки, нужно упаковать содержимое, которое необходимо прокручивать, в ScrollViewer. Несмотря на то, что этот элемент управления может хранить что угодно, обычно он используется для упаковки контейнера компоновки. Например, выше был показан пример, в котором элемент Grid применялся для создания таблицы с тремя столбцами текста, тексто-

вых окон и кнопок. Для прокрутки элемента Grid потребуется лишь упаковать Grid в ScrollViewer, как это показано в следующей сокращенной разметке:

```
<ScrollViewer Name="scroller">
<Grid Margin="0,10,0,0" Focusable="False">
<Grid.RowDefinitions>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition Width="*" MinWidth="50" Max-
Width="800"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
<Label Grid.Row="0" Grid.Column="0" Margin="3"
           VerticalAlignment="Center">Home:</Label>
<TextBox Grid.Row="0" Grid.Column="1" Margin="3"
           Height="Auto"  VerticalAlignment="Center"></TextBox>
<Button Grid.Row="0" Grid.Column="2" Margin="3" Pad-
ding="2">Browse</Button>

…

</Grid>
</ScrollViewer>
```

Результат показан на рис. 28.

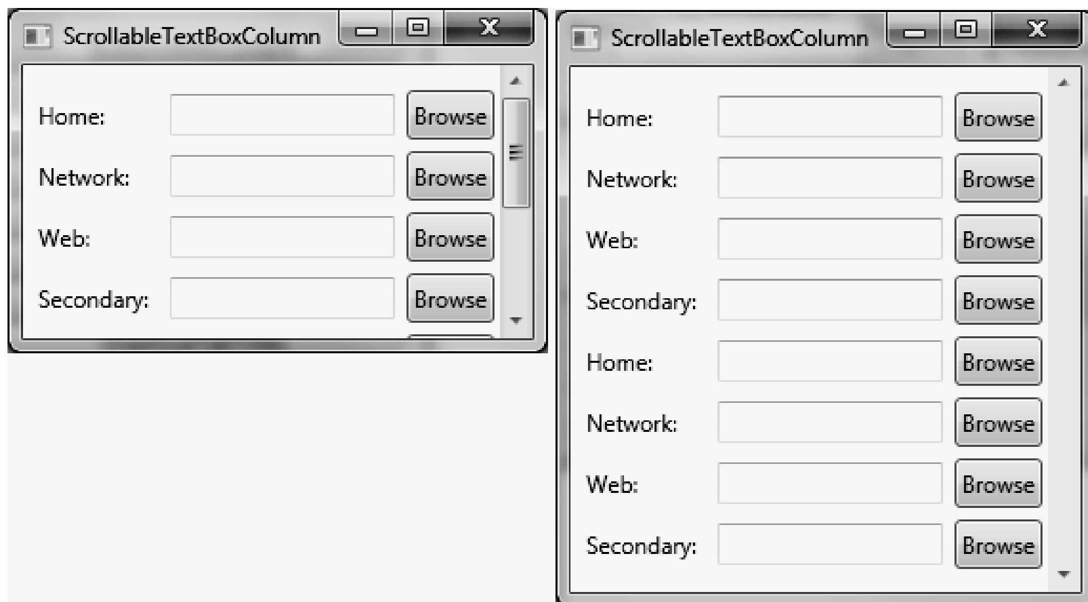

Рис. 27. Прокручивающееся окно и изменение его размера

Если в этом примере изменить размеры окна, чтобы оно могло уместить в себе все содержимое, полоса прокрутки станет неактивной, хотя ее по-прежнему можно будет видеть (см. правую часть рис. 28). Этим поведением можно управлять с помощью свойства VerticalScroll.BarVisibility, которое принимает значение из перечисления ScrollBarVisibility. В перечислении определены следующие значения:

- Значение Visible, используемое по умолчанию, задает вертикальную линейку прокрутки;
- Значение Auto необходимо для того, чтобы линейка прокрутки появлялась по мере необходимости и исчезала, если она не нужна;
- Значение Disabled используется в том случае, если линейку прокрутки вообще не нужно отображать.

Можно также использовать значение Hidden, действие которого похоже на Disabled, хоть и с некоторыми отличиями. Во-первых, при скрытой линейке прокрутки содержимое все равно можно прокручивать, например, с помощью клавиш управления курсором. Во-вторых, содержимое в элементе управления ScrollViewer размещается по-

другому. Когда присваивается значение Disabled, это означает, что содержимое будет занимать столько места, сколько его есть в элементе управления ScrollViewer. С другой стороны, если присваивается значение Hidden, содержимое будет занимать неограниченное пространство. Это означает, что содержимое может выйти за пределы области прокрутки. Обычно значение Hidden используется, когда нужно обеспечить другой механизм прокрутки, например, с помощью специальных кнопок для прокрутки. ScrollViewer поддерживает также горизонтальную прокрутку, хотя свойство HorizontalScrollBarVisibility по умолчанию имеет значение Hidden. Чтобы использовать горизонтальную прокрутку, нужно вместо этого значения указать Visible или Auto.

Чтобы прокрутить содержимое окна, показанного на рис. 28, можно щелкнуть на линейке прокрутки, перетащить ползунок, прокрутить колесико мыши, можно воспользоваться клавишей табуляции для перехода от одного элемента управления к другому, или можно щелкнуть где-нибудь на пустом месте в сетке и нажимать клавиши управления курсором. Если же этих возможностей недостаточно, можно использовать методы класса ScrollViewer для прокрутки содержимого программным способом:

- Наиболее очевидными методами являются LineUp() и LineDown(), которые эквивалентны щелчку на кнопках со стрелками, расположенных на вертикальной линейке прокрутки, что приводит к однократной прокрутке содержимого вверх или вниз;
- Также можно использовать методы PageUp() и PageDown(), которые позволяют прокручивать все содержимое на экране вниз или вверх, что равносильно щелчку на поверхности линейки прокрутки, выше или ниже ползунка;
- Похожие методы позволяют прокручивать содержимое по горизонтали: LineLeft(), LineRight(), PageLeft() и PageRight();

- Можно использовать методы ScrollToXXX(), чтобы перейти в какое-то определенное место. Для вертикальной прокрутки используются методы ScrollToEnd() и ScrollToHome(), которые переносят в заданную позицию. Существуют также и «горизонтальные» версии этих методов, к которым относятся ScrollToLeftEnd(), ScrollToRightEnd() и ScrollToHorizontalOffset().

На рис. 29 показан пример, в котором несколько специальных кнопок позволяют перемещаться по содержимому в элементе управления ScrollViewer. Каждая кнопка запускает простой обработчик события, который использует один из вышеупомянутых методов.

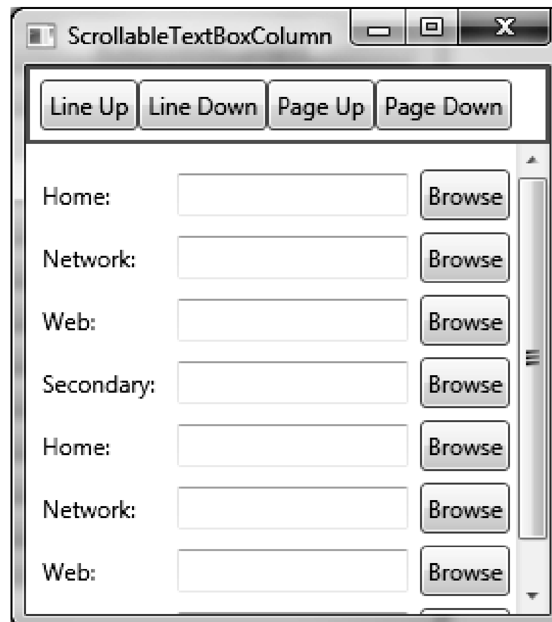

Рис. 28. Программная прокрутка

4.5.2. Специальная прокрутка

Встроенный вариант прокрутки элемента управления ScrollViewer является довольно полезным. Он позволяет плавно прокручивать любое содержимое, начиная со сложных векторных рисунков и заканчивая сеточными элементами. Однако одной из интригующих особенностей элемента управления ScrollViewer является возможность участия содержимого в процессе прокрутки. Продемонстрируем специальную прокрутку на примере:

1. Прокручиваемый элемент помещается внутрь элемента

управления ScrollViewer. Это может быть любой элемент, реализующий интерфейс IScrollInfo;

2. Сообщаем элементу управления ScrollViewer, что содержимое «знает» о способе прокрутки, присвоив свойству ScrollViewer.CanContentScroll значение true;
3. При взаимодействии с элементом управления ScrollViewer (посредством линейки прокрутки, колесика мыши, методов прокрутки и т. д.), он вызывает соответствующие методы при помощи интерфейса IScrollInfo. После этого элемент выполняет свою собственную прокрутку.

Интерфейс IScrollInfo реализуют всего несколько элементов. Одним из них является контейнер StackPanel. Его реализация интерфейса IScrollInfo реализует логическую прокрутку – прокрутку, которая осуществляет переход от элемента к элементу, а не от строки к строке.

Если поместить элемент управления StackPanel в ScrollViewer и не задать свойство CanContentScroll, то будет реализовано обычное поведение. При прокрутке вверх или вниз будет происходить перемещение одновременно нескольких пикселей. А если свойству CanContentScroll присвоить значение true, то при каждом щелчке будет осуществляться переход к началу следующего элемента (см. рис. 30):

```
<ScrollViewer CanContentScroll="True">
<StackPanel>
<Button Height="100">1</Button>
<Button Height="100">2</Button>
<Button Height="100">3</Button>
<Button Height="100">4</Button>
</StackPanel>
</ScrollViewer>
```

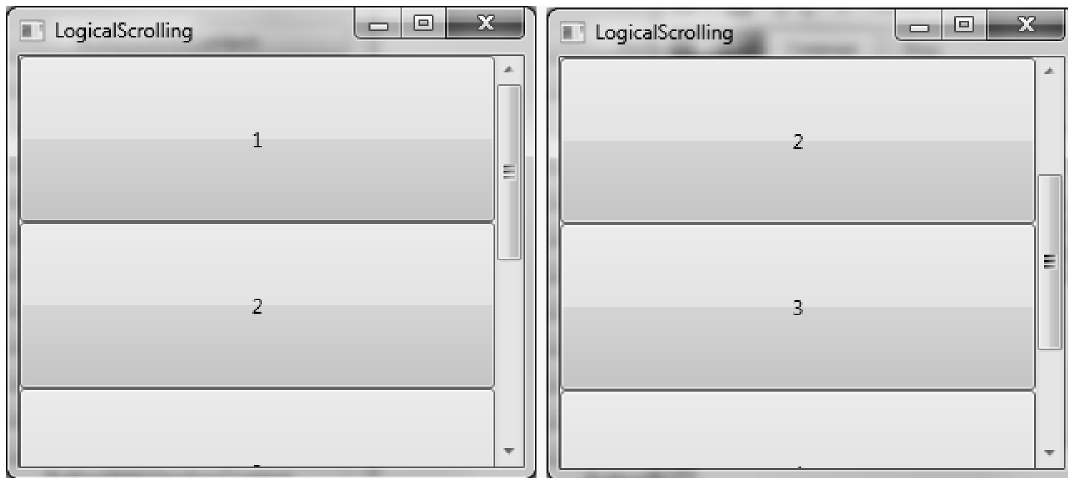

Рис. 29. Логическая прокрутка содержимого StackPanel

4.6. Элементы управления содержимым с заголовком

Одним из наследников класса ContentControl является класс HeaderedContentControl. Он представляет контейнер, который может иметь как одноэлементное содержимое, так и одноэлементный заголовок (хранится в свойстве Header).

У класса ContentControl есть три наследника: GroupBox, TabItem и Expander. Элемент управления GroupBox является самым простым из них. Он отображается ввиде окна со скругленными углами и заголовком. Ниже показан его пример (см. рис. 31):

```
<GroupBoxHeader="AGroupBoxTest" Padding="5"
Margin="5" VerticalAlignment="Top">
<StackPanel>
<RadioButton Margin="3">One</RadioButton>
<RadioButton Margin="3">Two</RadioButton>
<RadioButton Margin="3">Three</RadioButton>
<Button Margin="3">Save</Button>
</StackPanel>
</GroupBox>
```

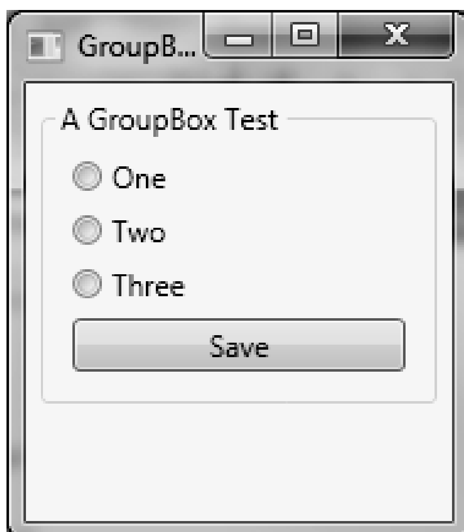

Рис. 30. Элемент управления GroupBox

Отметим, что для элемента управления GroupBox необходим контейнер (например, StackPanel), который поможет упорядочить его содержимое. GroupBox часто используется для группирования небольших наборов связанных элементов управления, таких как переключатели. Однако этот элемент управления не имеет встроенных функций, поэтому его можно применять для любых целей.

TabItem представляет страницу в элементе управления TabControl. Класс TabItem добавляет одно важное свойство IsSelected, которое показывает, отображается ли в данный момент вкладка в элементе управления TabControl. Ниже представлена разметка, необходимая для того, чтобы создать простой пример, показанный на рис. 32:

```
<TabControl Margin="5">
<TabItem Header="Tab One">
<StackPanel Margin="3">
<CheckBox Margin="3">Setting One</CheckBox>
<CheckBox Margin="3">Setting Two</CheckBox>
<CheckBox Margin="3">Setting Three</CheckBox>
</StackPanel>
</TabItem>
<TabItem Header="Tab Two"></TabItem>
</TabControl>
```

Рис. 31. Элемент управления TabControl

Как и свойство Content, свойство Header может принимать любой тип объекта. Оно отображает классы-наследники UIElement, визуализируя их и используя метод ToString() для внутристрочного текста и всех других объектов. Это означает, что можно создать групповое окно или вкладку с графическим содержимым или произвольными элементами в заголовке. Нижепоказанпример (см. рис. 33):

```
<TabControl Margin="5">
<TabItem>
<TabItem.Header>
<StackPanel>
<TextBlock Margin="3" >Image and Text Tab Title</TextBlock>
<Image Source="happyface.jpg" Stretch="None" />
</StackPanel>
</TabItem.Header>
<StackPanel Margin="3">
<CheckBox Margin="3">Setting One</CheckBox>
<CheckBox Margin="3">Setting Two</CheckBox>
<CheckBox Margin="3">Setting Three</CheckBox>
</StackPanel>
</TabItem>
```

```
<TabItem Header="Tab Two"></TabItem>
</TabControl>
```

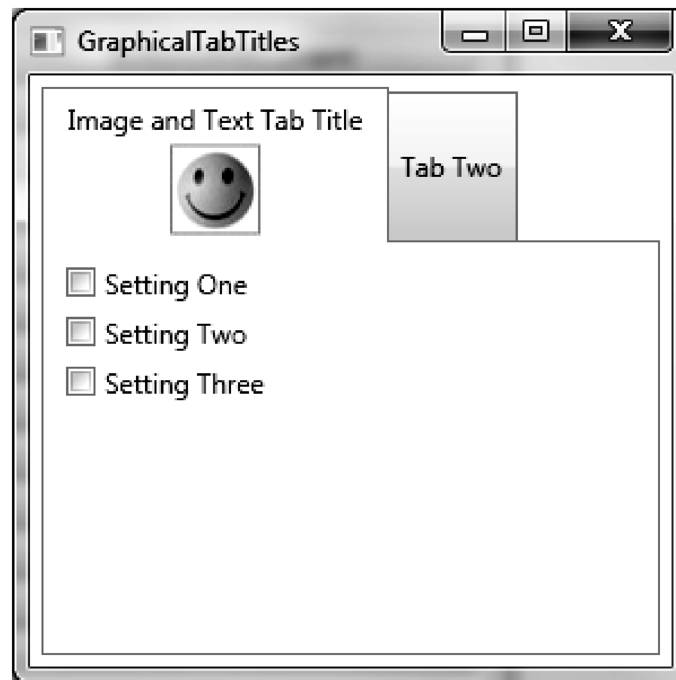

Рис. 32. Элемент TabControl

4.7. Элемент управления Expander

Самым оригинальным элементом управления содержимым является Expander. Он упаковывает область содержимого, которую пользователь может показывать или скрывать, щелкая на небольшой кнопке со стрелкой. Эта технология используется часто в оперативных справочных системах, а также на Web-страницах, чтобы они могли включать большие объемы содержимого, не перегружая пользователей информацией, которую им не хочется видеть.

На рис. 34 показано два представления окна с тремя расширителями. В версии, показанной слева, все три расширителя свернуты. В версии, показанной справа, два расширителя развернуты.

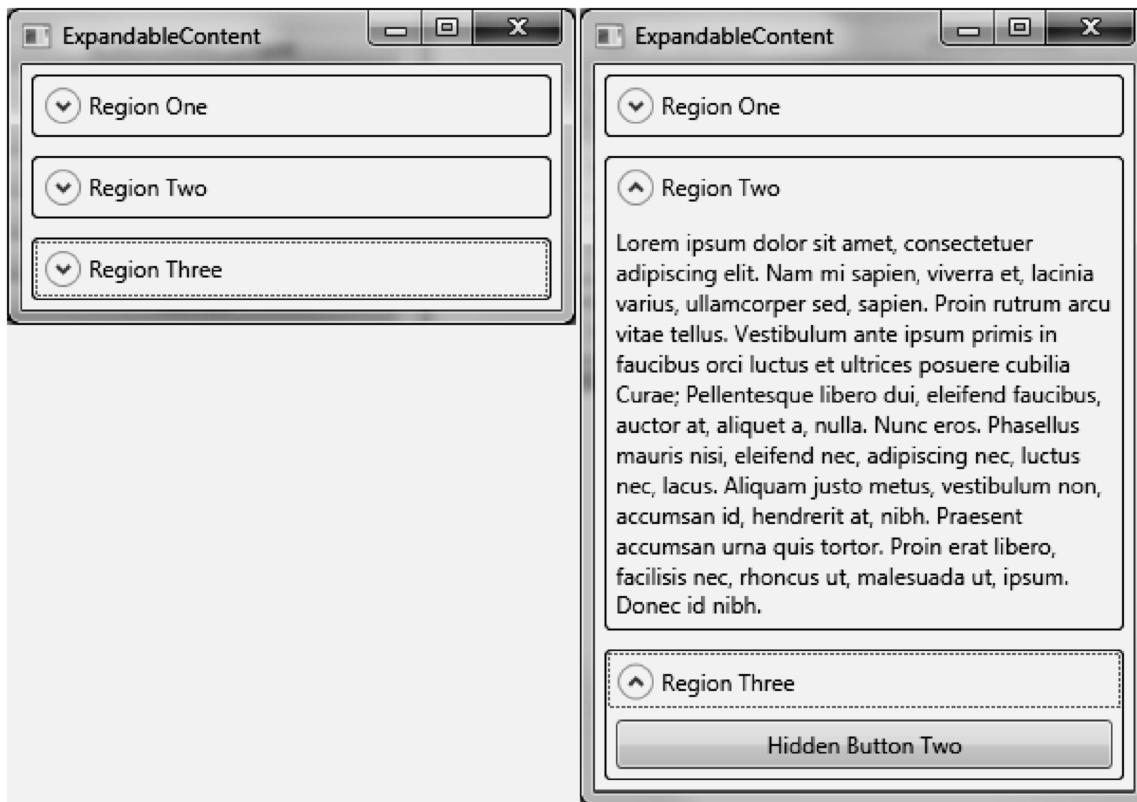

Рис. 33. Элемент Expander и скрытие элементов

Для применения элемента Expander нужно упаковать содержимое, которое необходимо сделать разворачивающимся. Как правило, каждый элемент управления Expander сначала находится в свернутом состоянии, однако это можно изменить в разметке (или в коде), установив свойство IsExpanded. Ниже показана разметка, создающая пример, представленный на рис. 34:

```
<StackPanel>

<ExpanderMargin="5" Padding="5" Header="RegionOne"
BorderThickness="1" BorderBrush="Black">
<Button Padding="3">Hidden Button One</Button>
</Expander>
<Expander Margin="5" Padding="5" Header="Region Two"
            BorderThickness="1" BorderBrush="Black">
<TextBlock TextWrapping="Wrap">
        Lorem ipsum dolor sit amet, consectetuer
…
```

```
nec, rhoncus ut, malesuada ut, ipsum. Donec id nibh.
</TextBlock>
</Expander>
<Expander Margin="5" Padding="5" Header="Region Three" IsEx-
panded="True"
              BorderThickness="1" BorderBrush="Black">
<Button Padding="3">Hidden Button Two</Button>
</Expander>
</StackPanel>
```

Также можно также выбрать направление, в котором будет развертываться расширитель. На рис. 34 используется значение по умолчанию (Down), хотя можно присвоить свойству ExpandDirection значение Up, Left или Right. Если расширитель свернут, стрелки всегда будут указывать на направление, в котором он будет разворачиваться.

Поведение элемента управления Expander можно разнообразить за счет использования разных значений свойства ExpandDirection, потому что результат в остальной части пользовательского интерфейса будет зависеть от типа контейнера. Некоторые контейнеры, такие как WrapPanel, просто растягивают другие элементы. Другие, подобные Grid, используют пропорциональную или автоматическую подгонку размеров. На рис. 35 показан пример сетки, насчитывающей четыре ячейки, с разными направлениями расширения.

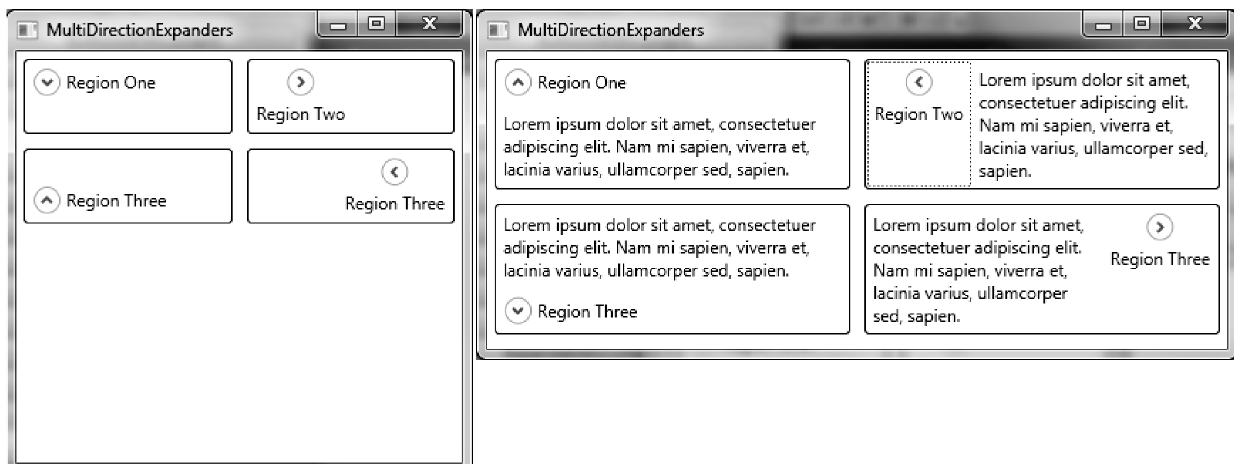

Рис. 34. Разворачивание в разных направлениях

Элемент управления Expander особенно подходит для использования в WPF, так как WPF основана на применении модели потоковой компоновки, которая может с легкостью обрабатывать области содержимого, растягивающиеся или сокращающиеся динамическим образом.

Если необходимо синхронизировать другие элементы управления с элементом управления Expander, то для этой цели следует обрабатывать события Expanded и Collapsed, которые возникают как раз перед тем, как содержимое появляется или исчезает. Благодаря этому можно реализовать так называемую отложенную или «ленивую» загрузку. Например, если процесс создания содержимого в элементе управления Expander является слишком затратным, можно подождать до тех пор, пока оно не будет показано, и только затем можно извлечь его. Или, возможно, есть необходимость обновить содержимое перед тем, как оно будет показано. В любом случае, можно реагировать на событие Expanded для выполнения необходимой работы.

Как правило, при разворачивании Expander его размеры увеличиваются, чтобы он мог уместить все содержимое. При этом, однако, может возникнуть проблема, если окно не является достаточно большим, чтобы оно могло уместить все содержимое при развертывании. С этой проблемой можно справиться благодаря нескольким стратегиям:

- Можно задать минимальный размер окна, чтобы оно могло уместить все содержимое, даже если окно будет иметь самые маленькие размеры;
- Можно задать свойство SizeToContent окна, чтобы окно развертывалось автоматически при открытии или закрытии Expander. Как правило, свойство SizeToContent имеет значение Manual, однако можно использовать значения Width или Height, чтобы развернуть его или свернуть до любого размера, достаточного для того, чтобы уместить содержимое;
- Можно ограничить размеры Expander, жестко закодировав его

свойства Height и Width. К сожалению, это наверняка приведет к усечению содержимого, если оно окажется слишком большим;

- Можно создать прокручиваемую и разворачиваемую область с помощью элемента управления ScrollViewer.

Фактически, эти технологии являются довольно простыми. Единственное, что требует дальнейшего разъяснения – это комбинированное использование элементов управления Expander и ScrollViewer. Чтобы этот подход мог работать, нужно жестко закодировать размеры элемента управления ScrollViewer. В противном случае он будет просто развернут, чтобы уместить его содержимое.

Ниже показан соответствующий пример (см. рис. 36):

```
<Expander Margin="5" Padding="5" Header="Region Two" >
<ScrollViewer Height="50" >
<TextBlock TextWrapping="Wrap">
…
</TextBlock>
</ScrollViewer>
</Expander>
```

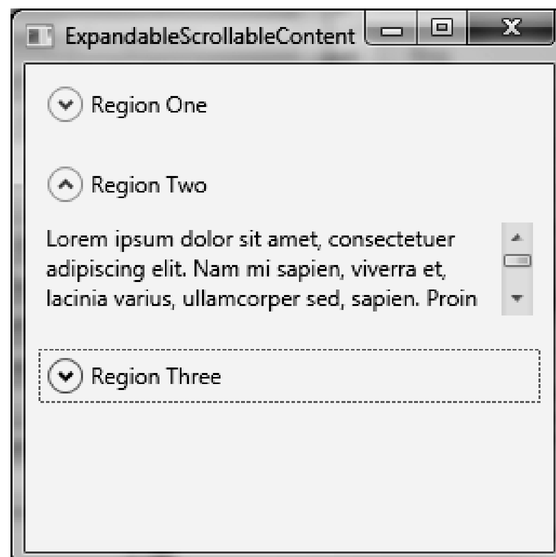

Рис. 36. Скроллирование содержимого элемента Expander

4.8. Декораторы

Выше рассматривались контейнеры, предназначенные для управления другими частями содержимого. В этом разделе будет показана другая ветвь элементов, подобных контейнерам, которые не являются элементами управления содержимым. Речь идет о декораторах, которые обычно служат для того, чтобы графически разнообразить и украсить область вокруг объекта.

Все декораторы являются наследниками класса System.Windows.Controls.Decorator. Большинство декораторов предназначено для использования вместе с определенными элементами управления. Например, элемент управления Button применяет декоратор ButtonChrome, чтобы создать свой характерный скругленный угол и затененный фон, а элемент управления ListBox использует декоратор ListBoxChrome. Существуют еще два общих декоратора, применять которые имеет смысл при создании пользовательских интерфейсов: Border и Viewbox.

4.8.1. Декоратор Border

Класс Border очень прост. Он принимает отдельную порцию вложенного содержимого (которым часто является панель компоновки) и добавляет к нему фон или рамку.

Для управления декоратором Border предлагаются свойства, перечисленные в таблице 6.

Таблица 6

Свойства класса Border

Имя	Описание
Background	Задает фон, который отображается за всем содержимым в рамке с помощью объекта Brush.
BorderBrush и BorderThickness	Эти свойства задают цвет рамки, которая отображается по краю объекта Border, используя объект Brush, и ширину рамки. Для отображения рамки нужно задать оба свойства.
CornerRadius	Позволяет закруглить углы рамки. Чем больше значение CornerRadius, тем более выразительным будет эффект закругления.
Padding	Добавляет пустое пространство между рамкой и содержимым, находящимся внутри.

Ниже показан пример простой рамки со слегка скругленными краями вокруг кнопок в контейнере StackPanel:

```
<Border Margin="5" Padding="5" Background="LightYellow"
          BorderBrush="SteelBlue" BorderThickness="3,5,3,5"
CornerRadius="3"
          VerticalAlignment="Top">
<StackPanel>
<Button Margin="3">One</Button>
<Button Margin="3">Two</Button>
<Button Margin="3">Three</Button>
</StackPanel>
</Border>
```

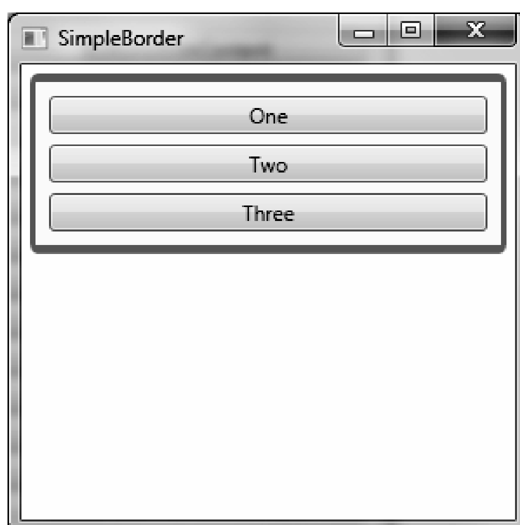

Рис. 35. Использование декоратора Border

Результат показан на рис. 37.

4.8.2. Декоратор Viewbox

Viewbox является более оригинальным декоратором. Любое содержимое, которое помещается в декоратор Viewbox, масштабируется таким образом, чтобы оно могло уместиться в этом декораторе.

Процесс масштабирования, выполняемый декоратором Viewbox, более изощренный, чем настройка параметров выравнивания, о которых речь шла выше. При растягивании элементапроисходит простое изменение пространства, доступного для данного элемента. Это изме-

нение не будет иметь никакого эффекта для большей части векторного содержимого, поскольку при рисовании векторов обычно используются фиксированные координаты.

Например, рассмотрим примеркнопки с формами, которая была показана выше. Эта форма помещается в элемент Grid, который подгоняет свои размеры, чтобы уместить все многоугольники. Если придать кнопкебольшие размеры, форма не изменится – она просто будет центрирована внутри кнопки (см. рис. 38). Это связано с тем, что размер каждого многоугольника задается в абсолютных координатах.

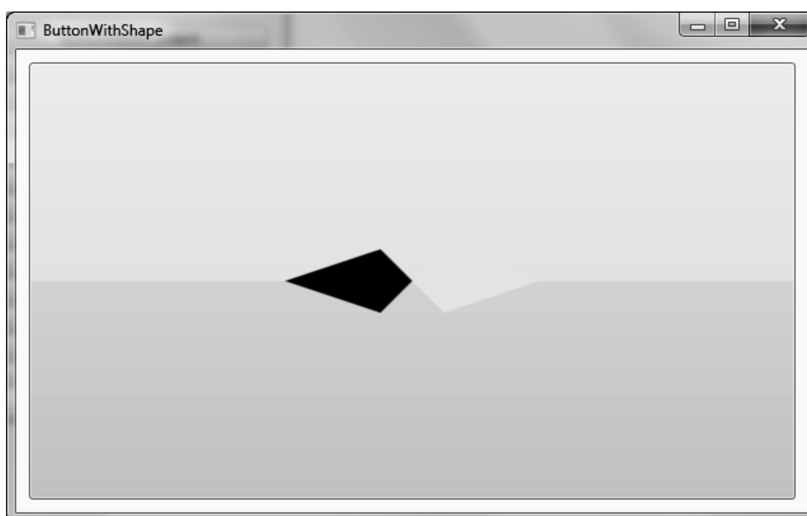

Рис. 36. Графическая кнопка с измененными размерами

Масштабирование, выполняемое декоратором Viewbox. подобно масштабированию, которое можно встретить в WPF, если увеличить системные настройки DPI. При масштабировании пропорционально изменяется каждый элемент экрана, включая изображения, текст, линии и формы, а также рамки обычных элементов, таких как кнопки. Если вернуться к примеру с кнопкой и формами, и поместить сетку Grid в декоратор Viewbox, получится результат, показанный на рис. 39:

```
<Button Margin="10">
<Viewbox>
<Grid>
<Polygon Points="100,25 125,0 200,25 125,50"
             Fill="Yellow" />
```

```
<Polygon Points="100,25 75,0 0,25 75,50"
             Fill="Purple"/>
</Grid>
</Viewbox>
</Button>
```

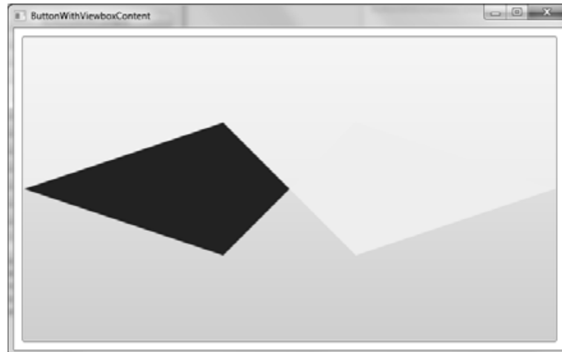

Рис. 37. Кнопка с фигурой в декораторе Viewbox

Несмотря на то, что многоугольники в Grid используют жестко закодированные координаты, Viewbox знает, как их нужнопреобразовывать. Способ преобразования координат выбирается путем сравнения требуемых размеров Grid идоступного размера. Например, если Viewbox в два раза больше требуемого размера Grid, то Viewbox масштабирует все свое содержимое с коэффициентом 2.

По умолчанию Viewbox выполняет пропорциональное масштабирование, которое сохраняет коэффициент пропорциональности своего содержимого. Т. е., даже если будет изменена форма кнопки, форма внутри не изменится. Однако это поведение можно изменить свойством Viewbox.Stretch. По умолчанию оно имеет значение Uniform. Если ему присвоить значение Fill, содержимое внутри декоратора Viewbox будет растянуто в обоих направлениях, чтобы занять все доступное пространство, даже если будут потеряны пропорции первоначального рисунка. Кроме этого, можно использовать свойство StretchDirection. По умолчанию оно имеет значение Both, однако если ему присвоить значение UpOnly, то содержимое будет растянуто только вверх, а не поширине, а если значение DownOnly – то содержимое будет растянуто только по ширине, а не вверх.

5. СВОЙСТВА ЗАВИСИМОСТЕЙ И МАРШРУТИЗИРУЕМЫЕ СОБЫТИЯ

Каждый программист, работающий с .NET, знаком со свойствами и событиями, которые являются основными компонентами объектной абстракции .NET. Появление WPF привело к изменению этих компонентов.

Во-первых, вместо обычных свойств .NET в WPF появилось средство более высокого уровня – свойства зависимостей (dependency property). Свойства зависимостей гораздо эффективнее потребляют память и поддерживают такие высокоуровневые возможности, как уведомление об изменениях и наследование значений свойств (это способность распространять значения, используемые по умолчанию, вниз по дереву элементов). Свойства зависимостей являются также основой для определенного количества ключевых возможностей WPF, к числу которых можно отнести анимацию, привязку данных и стили. Однако, несмотря на изменения, заложенные в самой основе, в коде по-прежнему можно считывать и устанавливать свойства зависимостей точно так же, как и при использовании традиционных свойств .NET.

Во-вторых, вместо обычных событий .NET стали применяться события более высокого уровня – маршрутизируемые события (routedevent). Маршрутизируемые события – это события, которые обладают большими возможностями перемещения. Их суть простая – они могут спускаться или подниматься по дереву элементов, и по ходу своего путешествия попадают к обработчикам событий. Маршрутизируемые события позволяют выполнять обработку события в одном элементе, несмотря на то, что это событие может возникнуть в совершенно другом элементе. Как и свойства зависимостей, маршрутизируемые события могут использоваться обычным способом – путем подключения к обработчику событий, имеющему правильную сигнатуру. Кратко рассмотрим принципы их работы.

5.1. Маршрутизированные события

Каждый разработчик, использующий .NET, знаком с понятием-события – это сообщение, которое посылается объектом (например, элементом WPF), чтобы уведомить код о том, что что-то произошло. WPF улучшает модель событий .NET благодаря новой концепции-маршрутизации событий. Маршрутизация позволяет событию возникать в одном элементе, а генерироваться – в другом. Например, маршрутизация событий позволяет щелчку, начавшемуся в кнопке панели инструментов, генерироваться в панели инструментов, а затем во вмещающем панель окне, и только потом обрабатываться кодом.

Маршрутизация событий предлагает большую гибкость для написания лаконичного кода, который сможет обрабатывать события в более удобном для этого месте. Она необходима также для работы с моделью содержимого WPF, которая позволяет создавать простые элементы (например, кнопки) из десятков отдельных компонентов, каждый из которых имеет свой собственный набор событий.

5.1.1. Присоединение обработчика событий

Как было показано выше, присоединить обработчик события можно несколькими способами. Чаще всего для этой цели добавляется атрибут события в разметку XAML. Данный атрибут события получает имя события, которое необходимо обрабатывать, а его значение получает имя метода обработчика события. Ниже показан пример, в котором этот синтаксис применяется для соединения события MouseUp элемента Image с обработчиком события img_MouseUp:

```
<Image Source="happyface.jpg" Stretch="None"
MouseUp="SomethingClicked" />
```

Хотя это и не обязательно, обычно имя метода обработчика события задается в виде ИмяЭлемента_ИмяСобытия. Если элемент не имеет определенного имени (возможно, по причине того, что нет необходимости взаимодействовать с ним в любом другом месте в коде), можно использовать имя, которое он мог бы иметь:

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

Также можно соединить событие с кодом. Ниже приведен эквивалент кода разметки XAML, показанной выше:

```
img.MouseUp += new MouseButtonEventHandler(img_MouseUp);
```

Этот код создает объект делегата, имеющий правильную сигнатуру для события (в данном случае это экземпляр делегата MouseButtonEventHandler) и указывающий метод img_MouseUp(). Затем он добавляет делегата в список зарегистрированных обработчиков событий для события img.MouseUp.Язык C# разрешает применять более рациональный синтаксис, явным образом создающий подходящий объект делегата: `img.MouseUp += img_MouseUp;`

Подход с использованием кода будет полезным, если нужно динамически создавать элемент управления и присоединять обработчик события в некоторой точке в течение времени существования окна. События, которые включаются в XAML, всегда присоединяются при первом создании экземпляра объекта окна. Этот подход позволяет также упростить и рационализировать код XAML, что будет исключительно полезным, если планируется совместно использовать его не с программистами, а, например, с художниками-дизайнерами. Недостатком является увеличение количества строк кода, который увеличит объем файлов с кодом.

Подход, продемонстрированный в предыдущем коде, основан на упаковщике события, который вызывает метод UIElement.AddHandler(). Также можно связать событие напрямую, самостоятельно вызвав метод UIElement.AddHandler (). Ниже показан пример:

```
img.AddHandler(Image.MouseUpEvent,new MouseButtonEventHand-
ler(img_MouseUp));
```

Если используется этот подход, то придется создавать подходящий тип делегата (например, MouseButtonEventHandler). Нельзя создать объект делегата неявно, как это делается при подключении со-

бытия через упаковщик свойства. Это объясняется тем, что метод UIElement.AddHandler() поддерживает все события WPF и не знает о том, какой тип делегата нужно использовать.

Некоторые разработчики предпочитают использовать имя класса, в котором определено событие, а не имя класса, сгенерировавшего событие. Ниже показан эквивалентный синтаксис, наглядно демонстрирующий, как событие MouseUpEvent определено в UIElement.

```
img.AddHandler(UIElement.MouseUpEvent, new
MouseButtonEventHandler(img_MouseUp));
```

Если нужно отсоединить обработчик события, то единственным решением в этом случае является написание соответствующего кода. Для этого можно воспользоваться операцией, показанной ниже:

```
img.MouseUp -= img_MouseUp;
```

Или же можно вызвать метод UIElement.RemoveHandler ():

```
img.RemoveHandler(Image.MouseUpEvent, new
MouseButtonEventHandler(img_MouseUp));
```

5.2. Маршрутизация событий

Как упоминалось выше, многие элементы управления в WPF являются элементами управления содержимым, которые могут иметь разный тип и разный объем вложенного содержимого. Например, можно создать графическую кнопку без формы, создать метку, которая будет совмещать текст и рисунки, или поместить содержимое в специальный контейнер, чтобы его можно было прокручивать или разворачивать окно во весь экран. Можно даже повторять процесс «вкладывания» необходимое количество раз.

При этом возникает следующий вопрос. Например, предположим, что имеется метка, в которой содержится панель StackPanel, и вместе они формируют два блока текста и изображения:

```
<Label BorderBrush="Black" BorderThickness="1">
<StackPanel>
<TextBlock Margin="3"> Image and text label</TextBlock>
```

```
    <Image Source="happyface.jpg" Stretch="None" /><TextBlock
    Margin="3"> Courtesy of the StackPaneI
</TextBlock>
</StackPanel>
</Label>
```

Как было сказано выше, каждый ингредиент, помещаемый в окно WPF, является наследником класса UIElement, включая Label, StackPanel,TextBlockиImage. Класс UIElement определяет несколько ключевых событий. Например, каждый класс, являющийся потомком UIElement, предлагает события MouseUp и MouseDown.

А теперь посмотрим, что произойдет, если щелкнуть на изображении в данной метке. Понятно, что при этом возникнут события Image.MouseDown и Image.MouseUp. Но что делать, если нужно обрабатывать все щелчки на метке одинаковым образом? В этом случае не будет разницы в том, где щелкнул пользователь – на изображении, на тексте или на пустом месте в области метки. В любом из этих случаев необходимо реагировать спомощью одного и того же кода. Очевидно, что можно привязать один и тот же обработчик к событиям MouseDown и MouseUp каждого элемента, однако это может запутать код и усложнить сопровождение разметки. WPF предлагает более удобное решение за счет модели маршрутизируемых событий.

Маршрутизируемые события бывают трех видов:

- Прямые события (directevent) подобны обычным событиям .NET. Они возникают в одном элементе, и не передаются в другой. Например, MouseEnter является простым событием;
- Поднимающиеся («пузырьковые») события (bubbling event) перемещаются вверх по иерархии. Например, MouseDown является поднимающимся событием. Оно возникает в элементе, на котором был произведен щелчок. Затем оно передается от этого элемента к родителю, затем к родителю этого родителя, и так далее. Этот процесс продолжается до тех пор, пока WPF не достигнет вершины дерева элементов;

- Туннельные события (tunneling event) перемещаются вниз по иерархии. Они дают возможность предварительно просматривать и, возможно, останавливать событие до того, как оно дойдет до подходящего элемента управления. Например, PreviewKeyDown позволяет прервать нажатие клавиши сначала на уровне окна, а затем в более специфических контейнерах, до тех пор, пока не будет достигнут элемент, который имел фокус ввода в момент нажатия клавиши.

Поскольку события MouseUp и MouseDown являются поднимающимися событиями, теперь можно определить, что произойдет в примере с меткой. При щелчке на лице с улыбкой событие MouseDown возникнет в следующем порядке:

1. Image.MouseDown;
2. StackPanel.MouseDown;
3. Label.MouseDown.

После того как событие MouseDown возникнет в метке, оно пройдет до следующего элемента управления (которым в этом случае является сетка Grid, разбивающая вмещающее окно), а затем до его родителя (окно). Окно находится на самом верху иерархии и в самом конце в последовательности поднятия события. Здесь есть последняя возможность обработать поднимающееся событие, такое как MouseDown. Если пользователь отпускает кнопку мыши, событие MouseUp возникает в такой же последовательности.

Обрабатывать поднимающиеся события можно не только в одном месте. В действительности, события MouseDown и MouseUp можно обрабатывать на любом уровне. Однако, как правило, для этой задачи выбирается наиболее подходящий на данный момент уровень.

5.2.1. Класс RoutedEventArgs

Когда обрабатывается поднимающееся событие, параметр отправителя содержит ссылку на последнее звено в цепочке. Например,

если событие поднимается вверх от изображения до метки, прежде чем произойдет его обработка, параметр отправителя будет ссылаться на объект метки.

В некоторых случаях нужно будет определить, где первоначально произошло событие. Эту информацию, а также другие подробности, можно получить в свойствах класса RoutedEventArgs, перечисленных в таблице 7. Поскольку все классы аргументов событий WPF являются наследниками RoutedEventArgs, эти свойства доступны в любом обработчике события.

Таблица 7

Свойства класса RoutedEventArgs

Имя	Описание
Source	Показывает, какой объект сгенерировал событие. Если речь идет о событии клавиатуры, то этим объектом будет элемент управления, находившийся в фокусе в момент возникновения события. Если это событие мыши, то этим объектом будет самый верхний элемент под указателем мыши в момент возникновения события.
OriginalSource	Показывает, какой объект сгенерировал событие. Как правило, OriginalSource является тем же, что и источник. Однако в некоторых случаях OriginalSource спускается глубже по дереву объектов, чтобы дойти до декоратора элемента, являющегося частью элемента более высокого уровня. Например, если щелкнуть кнопкой мыши, чтобы закрыть рамку окна, то в качестве источника события будет объект Window и Border в качестве исходного источника.
RoutedEvent	Предлагает объект RoutedEvent для события, сгенерированного обработчиком события. Эта информация будет полезной, если разные события обрабатываются с помощью одного и того же обработчика.
Handled	Позволяет остановить процесс поднятия или опускания события. Если свойство Handled элемента управления имеет значение true, событие не будет продолжать продвижение, и не будет возникать в любых других элементах.

5.3. Поднимающиеся события

На рис. 40 показано простое окно, в котором видно, как поднимается событие. При щелчке на какой-либо части метки, события будут возникать в порядке, перечисленном в окне списка, и приведен вид этого окна сразу после того, как пользователь щелкнул на изображении внутри метки. Событие MouseUp проходит пять уровней, останавливаясь на специальной форме BubbledLabelClick.

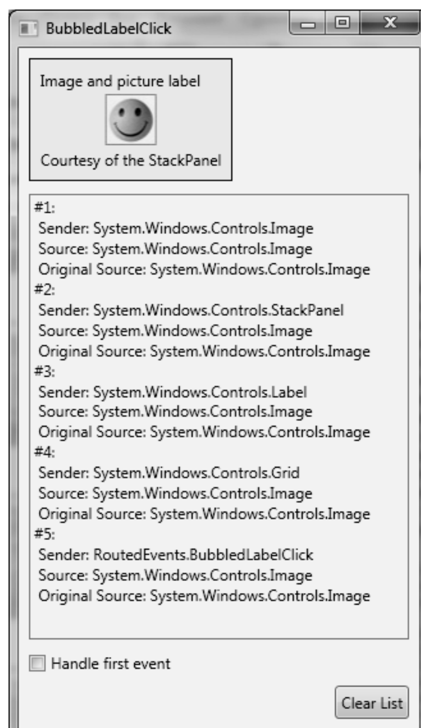

Рис. 38. Щелчок на изображении

Чтобы получить эту форму, нужно связать изображение и каждый элемент, стоящий над ним в иерархи элементов, с одним и тем же обработчиком события – методом SomethingClicked(). Ниже показано, как это делается средствами XAML:

```
<Windowx:Class="RoutedEvents.BubbledLabelClick"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="BubbledLabelClick" Height="359" Width="329"
    MouseUp="SomethingClicked"
>
```

```
<Grid Margin="3" MouseUp="SomethingClicked">
<Grid.RowDefinitions>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="*"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
<RowDefinition Height="Auto"></RowDefinition>
</Grid.RowDefinitions>
<Label Margin="5" Background="AliceBlue" BorderBrush="Black"
BorderThickness="1" MouseUp="SomethingClicked" HorizontalAlign-
ment="Left" >
<StackPanel MouseUp="SomethingClicked" >
<TextBlock Margin="3" MouseUp="SomethingClicked" >
            Image and picture label</TextBlock>
<Image Source="happyface.jpg" Stretch="None"
                 MouseUp="SomethingClicked" Image-
Failed="Image_ImageFailed" />
<TextBlock Margin="3"
                     MouseUp="SomethingClicked" >
            Courtesy of the StackPanel</TextBlock>
</StackPanel>
</Label>
<ListBox Margin="5" Name="lstMessages" Grid.Row="1"></ListBox>
<CheckBox Margin="5" Grid.Row="2" Name="chkHandle">Handle first
event</CheckBox>
<Button Click="cmdClear_Click" Grid.Row="3" HorizontalAlign-
ment="Right" Margin="5" Padding="3">Clear List</Button>
</Grid>
</Window>
```

Метод SomethingClicked() просто проверяет свойства объекта RoutedEventArgsи добавляет сообщение в окно списка:

```
protected int eventCounter = 0;
private void SomethingClicked(object sender, RoutedEventArgs e)
{
eventCounter++;
```

```
string message = "#" + eventCounter.ToString() + ":\r\n" +
" Sender: " + sender.ToString() + "\r\n" +
" Source: " + e.Source + "\r\n" +
" Original Source: " + e.OriginalSource;
lstMessages.Items.Add(message);
    e.Handled = (bool)chkHandle.IsChecked;
}
```

В этом примере есть еще одна деталь. Если отметить флажок chkHandle, метод SomethingClicked() присвоит свойству RoutedEventArgs.Handled значение true, в результате чего будет остановлена последовательность поднятия события в момент его возникновения. Поэтому в списке будет показано только первое событие, как на рис. 41.

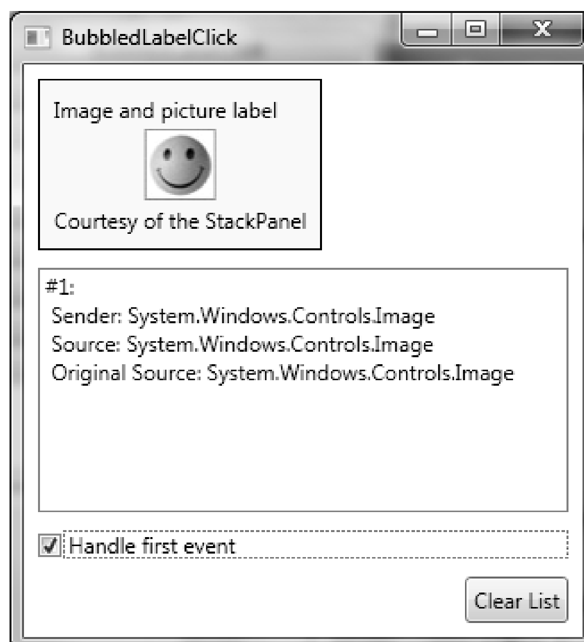

Рис. 39. Первое событие щелчка мыши

Поскольку метод SomethingClicked() обрабатывает событие MouseUp, которое возникает в объекте Window, имеется возможность перехватывать щелчки в окне списка и на пустой поверхности окна. Однако событие MouseUp не возникает при щелчке на кнопке Clear. Это связано с тем, что кнопка включает фрагмент кода, который блокирует событие MouseUp и генерирует событие более высокого уров-

ня – Click. Между тем, флагу Handled присваивается значение true, вследствие чего запрещается дальнейшее продвижение события MouseUp.

5.3.1. Обработка заблокированного события

Отметим, что можно получать события, которые отмечены как обработанные. Вместо того чтобы присоединять обработчик события посредством XAML, необходимо использовать для этой цели рассмотренный ранее метод AddHandler(). Этот метод предлагает перегрузку, которая принимает булевское значение в третьем параметре. Если этим значением будет true, событие будет получено, даже если для него был установлен флаг Handled:

```
cmdClear.AddHander(UIElement.MouseUpEvent,
new MouseButtonEventHandler(cmdClear_MouseUp), true);
```

Это очень хороший вариант решения. Кнопка предназначена для блокирования события MouseUp по очень простой причине: чтобы избежать возникновения конфликта. В конце концов, в Windows принято, что нажать кнопку с помощью клавиатуры можно несколькими способами. Если будет допущена ошибка, обработав событие MouseUp в элементе Button вместо события Click, код отреагирует только на щелчки мышью, а не на эквивалентные действия со стороны клавиатуры.

5.4. Прикрепляемые события

Пример декоративной метки является довольно простым примером поднятия события, поскольку все элементы поддерживают событие MouseUp. Тем не менее, многие элементы управления обладают собственными специальными событиями. Кнопка является одним из таких примеров – она добавляет событие Click, которое не определено ни в одном базовом классе.

Предположим, что в элемент StackPanel помещается набор кнопок. Необходимо обработать все щелчки накнопке в одном обработ-

чике события. Грубый подход предусматривает присоединение события Click к каждой кнопке в одном и том же обработчике события. Однако событие Click поддерживает поднятие событий, что позволит решить задачу более изящным способом. Например, можно обработать все щелчки на кнопке, обрабатывая событие Click на более высоком уровне (например, на уровне элемента StackPanel).

К сожалению, следующий код работать не будет:

```
<StackPanel Click="DoSomething" Margin="5">
<Button Name="cmdl">Command l</Button>
<Button Name="cmd2">Command 2</Button>
<Button Name="cmd3">Command 3</Button>
</StackPanel>
```

Дело в том, что StackPanel не включает событие Click, поэтому этот код вызовеет ошибку во время синтаксического анализа XAML. Для решения этой задачи нужно использовать другой синтаксис с применением присоединенных событий в виде ИмяКласса.ИмяСобытия. Ниже показан подходящий вариант:

```
<StackPanel Button.Click="DoSomething" Margin="5">
<Button Name="cmdl">Command l</Button>
<Button Name="cmd2">Command 2</Button>
<Button Name="cmd3">Command 3</Button>
</StackPanel>
```

Теперь обработчик события получит щелчок для всех кнопок, содержащихся в элементе StackPanel.

Также можно подключить прикрепляемое событие в коде, однако вместо обычной операции += придется использовать метод UIElement.AddHandler(). Ниже показан такой пример (предполагается, что элемент StackPanel имеет имя pnlButtons):

```
pnlButtons.AddHandler (Button.Click, newRoutedEventHandler (Do-
Something));
```

В обработчике события DoSomething() можно несколькими способами определить, какая кнопка сгенерировала событие. Например,

можно сравнить ее текст (этот способ может привести к проблемам с локализацией) или ее имя. Лучше всего проверить, было ли задано свойство Name каждой кнопки с помощью XAML, чтобы имелся доступ к соответствующему объекту посредством поля в классе окна и сравнить эту ссылку с отправителем события. Ниже показан пример:

```
private void DoSomething(object sender, RoutedEventArgs e)
{
if (sender == cmdl) { ... 1
else
if (sender == cmd2) { ... )
else
if (sender == cmd3) { ... }
}
```

Существует еще один вариант – вместе с кнопкой отправить порцию информации, которую можно использовать в коде. Например, можно задать свойство Tag каждой кнопки, как показано ниже:

```
<StackPanel Click="DoSomething" Margin="5">
<Button Name="cmdl" Tag="The first button.">Command l</Button>
<Button Name="cmd2" Tag="The second button.">Command 2</Button>
<Button Name="cmd3" Tag="The third button,">Command 3</Button>
</StackPanel>
```

После этого можно обращаться к свойству Tag в коде:

```
private void DoSomething(object sender, RoutedEventArgs e)
{
object tag = ((FrameworkElement)sender).Tag;
MessageBox.Show((string)tag);
}
```

5.4.1. Туннельные события

Туннельные события работают точно так же, как и поднимающиеся события, но в обратном направлении. Например, если бы событие MouseUp было туннельным, то при щелчке на метке событие MouseUp возникло бы сначала в окне, затем в элементе Grid, затем в

StackPanel и так далее до тех пор, пока не будет достигнут источник, которым является изображение в метке.

Туннельные события имеют приставку Preview. Более того, WPF обычно определяет поднимающиеся и туннельные события парами. Это означает, что если существует поднимающееся событие MouseUp, то, скорее всего, также существует туннельное событие PreviewMouseUp. Туннельное событие всегда возникает перед поднимающимся событием, как показано на рис. 42:

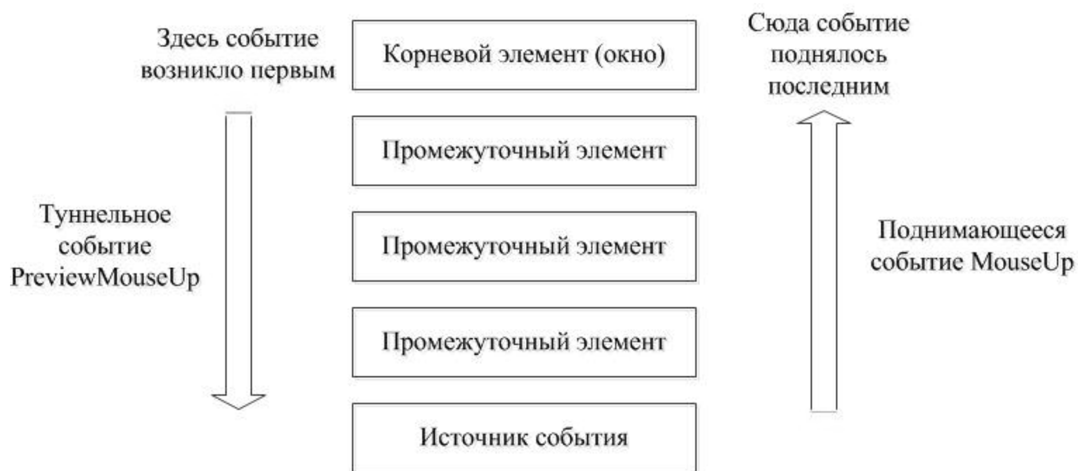

Рис. 40. Туннельные и поднимающиеся события

Также отметим, что если пометить туннельное событие как обработанное, событие поднятия не возникнет. Это связано с тем, что два события совместно используют один и тот же экземпляр класса RoutedEventArgs.

Туннельные события будут полезны, если необходимо выполнить некоторую предварительную обработку, связанную с некоторыми нажатиями клавиш, или отфильтровать некоторые события мыши. На рис. 43 показаны результаты проверки туннелирования на примере события PreviewKeyDown. Если нажать клавишу, находясь в текстовом поле, событие возникает сначала в этом поле, а затем спускается вниз по иерархии. А если на каком-то этапе пометить событие PreviewKeyDown как обработанное, то поднимающееся событие KeyDown не возникнет.

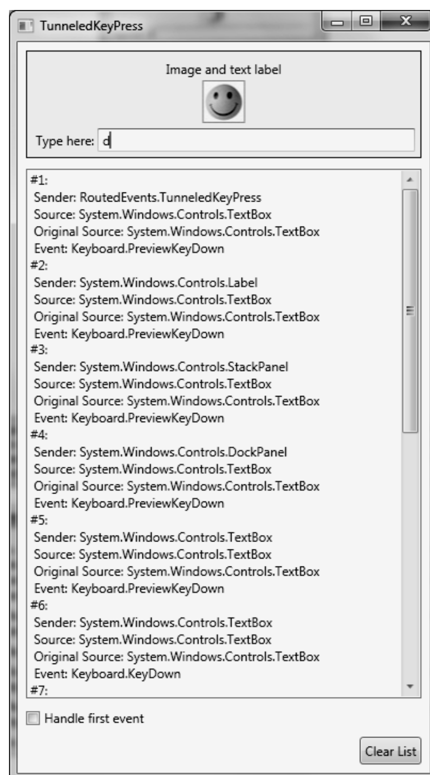

Рис. 41. Туннелирование нажатия клавиши

5.5. События WPF

Существуют самые разнообразные события, на которые можно реагировать в коде. Несмотря на то, что каждый элемент имеет широчайший спектр событий, наиболее важные события обычно делятся на четыре описанных ниже категории.

- События времени существования. Эти события возникают, когда элемент инициализируется, загружается или выгружается;
- События мыши. Эти события являются результатом действий мыши;
- События клавиатуры. Эти события являются результатом действий клавиатуры (например, нажатие клавиши);
- События пера. Эти события являются результатом использования пера, которое заменяет мышь в наладонных ПК.

Вместе события мыши, клавиатуры и пера известны как события ввода.

5.5.1. События времени существования

Все элементы генерируют события, когда они впервые создают-

ся и когда освобождаются. Эти события можно использовать для инициализации окна. События времени существования перечислены в таблице 8, все они определены в классе FrameworkElement.

Таблица 8

События времени существования

Имя	Описание
Initialized	Возникает после создания экземпляра элемента и определения его свойств в соответствии с правилами разметки XAML. В этот момент элемент инициализируется, хотя остальные части окна могут еще быть не инициализированными. Также на этом этапе еще не применены стили и привязка данных. Данное событие является обычным событием .NET.
Loaded	Возникает после того, как все окно было инициализировано и были применены стили и привязка данных. Это последний этап, после которого происходит собственно визуализация элемента. В этот момент свойство IsLoaded имеет значение true.
Unloaded	Возникает, когда элемент был освобожден. Это может быть вызвано или закрытием вмещающего окна, или удалением из окна специфического элемента.

Вмещающее окно имеет собственные события времени существования. Они перечислены в таблице 9.

Таблица 9

События времени существования класса Window

Имя	Описание
SourceInitialized	Возникает при запросе свойства HwndSource перед тем, как окно станет видимым. HwndSource – это дескриптор окна, который может понадобиться для вызова функций интерфейса Win32 API.
ContentRendered	Возникает сразу после первой визуализации окна. Этот момент не подходит для выполнения каких-либо изменений, которые могут повлиять на визуальное отображение окна, иначе придется выполнять вторую операцию по визуализации.
Activated	Возникает, когда пользователь переключается на это окно (например, из другого окна в приложении или вообще из другого приложения) или во время первой загрузки окна.
Deactivated	Возникает, когда пользователь уходит из этого окна (например, переходит в другое окно в приложении или вообще в другое приложение). Это событие возникает также, когда пользователь закрывает окно, после события Closing, но перед событием Closed.
Closing	Возникает при закрытии окна, что может быть вызвано или действием пользователя, или программой с помощью методов Window.Close() или Application.Shutdown(). Событие Closing дает возможность отменить операцию и оставить окно открытым, если присвоить свойству CancelEventArgs.Cancel значение true. Однако, если приложение завершит работу вследствие того, что пользователь выключил компьютер или вышел из системы, то сообщение получено не будет.
Closed	Возникает после закрытия окна. Это тот момент, когда объекты элемента по-прежнему остаются доступными, а событие Unloaded еще не возникло. В этот момент можно произвести очистку, записать параметры в файл и т. д.

Если нужно выполнить первичную инициализацию элементов управления, то наилучшим моментом для этого является событие Loaded. Как правило, в обработчике этого события можно выполнить все действия, связанные с инициализацией.

5.5.2. События ввода

Все события ввода – события, которые возникают вследствие действий мыши, клавиатуры или пера – передают дополнительную информацию в специальном классе аргументов событий. По сути, все эти классы совместно используют одного и того же предка – класс InputEventArgs. На рис. 44 показана иерархия наследования классов событий.

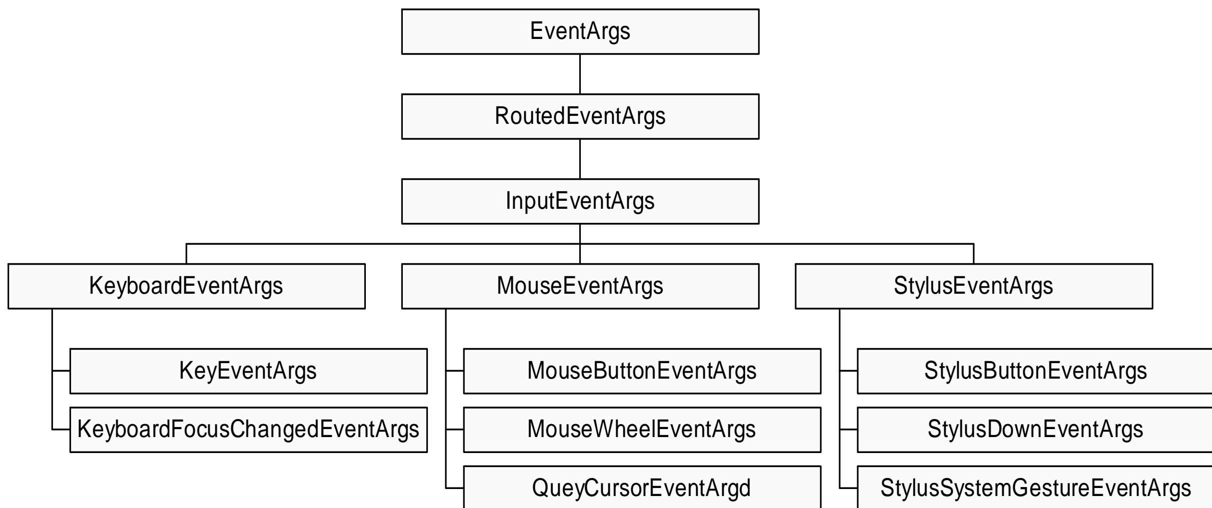

Рис. 42. Классы EventArgs для событий ввода

Класс InputEventArgs добавляет только два свойства: Timestamp и Device. Свойство Timestamp может иметь целочисленное значение, показывающее в миллисекундах, когда возникло событие. Свойство Device возвращает объект, который предлагает более подробную информацию об устройстве, сгенерировавшем событие, которым может быть мышь, клавиатура или перо. Каждый из этих трех вариантов представлен отдельным классом, и все они являются наследниками абстрактного класса System.Windows. Input.InputDevice.

5.5.3. Ввод с использованием клавиатуры

Когда пользователь нажимает клавишу, возникает целая серия событий. В таблице 10 перечислены события в порядке их возникновения.

Таблица 10

События клавиатуры в порядке возникновения

Имя	Тип маршрутизации	Описание
PreviewKeyDown	Туннелирование	Возникает при нажатии клавиши.
KeyDown	Поднятие	Возникает при нажатии клавиши.
PreviewTextInput	Туннелирование	Возникает, когда нажатие клавиши завершено, и элемент получает текстовый ввод. Это событие не возникает для тех клавиш, которые в результате не приводят к вводу текста.
TextInput	Поднятие	Возникает, когда нажатие клавиши завершено и элемент получает текстовый ввод. Это событие не возникает для тех клавиш, которые в результате не приводят к вводу текста.
PreviewKeyUp	Туннелирование	Возникает при отпускании клавиши.
KeyUp	Поднятие	Возникает при отпускании клавиши

Обработка событий, поступающих с клавиатуры, никогда не была столь легкой, как это может показаться. Некоторые элементы управления могут блокировать часть этих событий, поэтому они могут выполнять свою собственную обработку клавиатуры. Наиболее ярким примером является элемент TextBox, который блокирует событие TextInput. Элемент TextBox блокирует также событие KeyDown для некоторых нажатий клавиш, таких как клавиши управления курсором. В случаях, подобных этим, по-прежнему можно использовать туннельные события (PreviewTextInput и PreviewKeyDown).

Элемент управления TextBox добавляет одно новое событие – TextChanged. Это событие возникает сразу после того, как нажатие клавиши приводит к изменению текста в текстовом поле. В этот момент новый текст уже является видимым в текстовом поле, потому отменить нежелательное нажатие клавиши уже будет поздно.

5.5.4. Обработка нажатия клавиши

Чтобы понять, как работают и используются события клавиатуры, лучше всего рассмотреть пример. На рис. 45 показаноокно, которое следит за всеми возможными нажатиями клавиш, когда в фокусе находится текстовое поле, и реагирует на них, если они возникают. В окне показан результат ввода заглавной буквы S в текстовом поле.

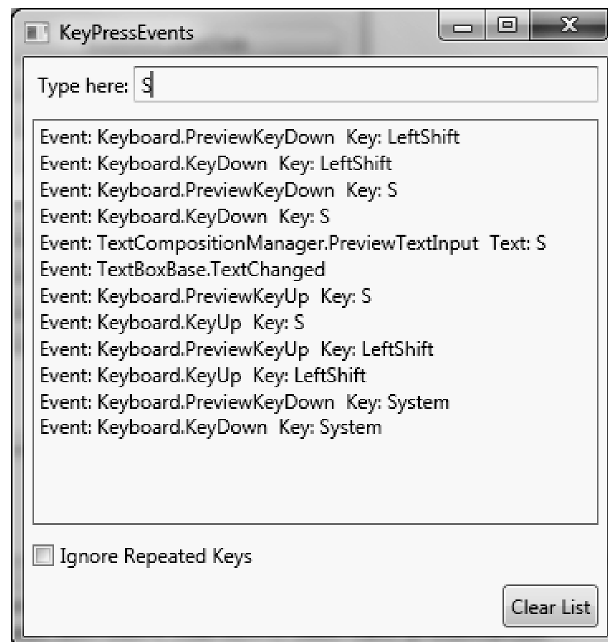

Рис. 43. Наблюдение за событиями клавиатуры

Этот пример демонстрирует один важный момент. События PreviewKeyDown и KeyDown возникают всякий раз, когда происходит нажатие клавиши. Однако событие TextInput возникает только тогда, когда в элементе был «введен» символ. Это действие на самом деле может включать нажатие многих клавиш.

В примере, показанном на рис. 45, нужно нажать две клавиши, чтобы получить заглавную букву S. Сначала необходимо нажать клавишу <Shift>, а затем клавишу <S>. Как результат, можно увидеть по два события KeyDown и KeyUp, и только одно событие Textlnput.

Каждое из событий PreviewKeyDown, KeyDown, PreviewKey и KeyUp дает одну и ту же информацию в объекте KeyEventArgs. Самой важной деталью является свойство Key, которое возвращает значение из перечисления System.Windows.Input.Key, показывающее

клавишу, которая была нажата или отпущена. Ниже представлен код обработчика события, который работает с событиями клавиш из примера, показанного на рис. 45:

```
private void KeyEvent(object sender, KeyEventArgs e)
{
if ((bool)chkIgnoreRepeat.IsChecked && e.IsRepeat) return;
string message = //"At: " + e.Timestamp.ToString() +
         "Event: " + e.RoutedEvent + " " +
" Key: " + e.Key;
lstMessages.Items.Add(message);
}
```

Значение Key не учитывает состояние любой другой клавиши. Например, нет разницы в том, была ли нажата клавиша <Shift> в тот момент, когда нажималась<S>; в любом случае, будет получено одно и то же значение Key, а именно Key.S.

В зависимости от того, какие параметры заданы для клавиатуры Windows, удержание клавиши в нажатом состоянии приводит к тому, что нажатие как действие повторяется через короткий промежуток времени. Например, удержание нажатой клавиши <S> приведет к вводу в текстовом поле целой серии символов S. Точно так же, удержание нажатой клавиши <Shift> приводит к повтору действия нажатия и к возникновению серии событий KeyDown. В реальном примере, в котором нажимается комбинация<Shift+S>, текстовое поле сгенерирует серию событий KeyDown для клавиши <Shift>, за ними – событие KeyDown для клавиши <S>, событие TextInput, а затем событие KeyUp для клавиш <Shift> и <S>. Если нужно проигнорировать повторы нажатия клавиши <S>, можно проверить, является ли нажатие результатом удерживания клавиши в нажатом состоянии, с помощью свойства KeyEventArgs. isRepeat, как показано ниже:

```
if ((bool)chkIgnoreRepeat.IsChecked && e.IsRepeat) return;
```

За событием KeyDown следует событие PreviewTextInput.

(Событие TextInput не возникает, поскольку элемент TextBox блокирует его.) В этот момент текст еще не отображается в элементе управления.

Событие TextInput предлагает код объекта TextCompositionEventArgs. Этот объект включает свойство Text, которое дает обработанный текст, подготовленный к получению элементом управления. Ниже представлен код, добавляющий текст в список, показанный на рис. 45:

```
private void TextInput(object sender, TextCompositionEventArgs
e)
{
string message = //"At: " + e.Timestamp.ToString() +
     "Event: " + e.RoutedEvent + " " +
" Text: " + e.Text;
lstMessages.Items.Add(message);
}
```

В идеале можно использовать PreviewTextInput для выполнения проверки в элементе управления, таком как TextBox. Например, если создается текстовое поле для ввода только чисел, можно проверить, не была ли введена при текущем нажатии клавиши буква; если буква не была введена, устанавливается флаг Handled. К сожалению, событие PreviewTextInput не генерируется для некоторыхклавиш, которые, возможно, придется обрабатывать. Например, если нажать клавишу пробела в текстовом поле, то событие PreviewTextInput будет пропущено. Это означает, что нужно будет обработать также событие PreviewKeyDown.

К сожалению, трудно реализовать надежную логику проверки данных в обработчике события PreviewKeyDown, поскольку все, чем можно оперировать – это значение Key, которое является слишком малой порцией информации. Например, перечисление Key проводит различие между цифровой клавиатурой и обычной клавиатурой. Это означает, что в зависимости от того, как будет нажата клавиша с циф-

рой 9, можно получить или значение Key.D9, или значение Key.NumPad9. Проверка всех дозволенных значений клавиши будет очень утомительной.

Одним из выходов является использование KeyConverter, чтобы преобразовать значение Key в более полезную строку. Например, использование функции KeyConverter.ConvertStringToString() в обоих значениях Key.D9 и Key.NumPad9 вернет результат «9» в форме строки. Если просто использовать преобразование Key.ToString(), можно получить менее полезное имя перечисления (либо «D9», либо «NumPad9»):

```
KeyConverter converter = newKeyConverter();
string key = converter.ConvertToString(e.Key);
```

Однако использовать KeyConverter тоже не очень неудобно, поскольку получается более объемный текст (например, «Backspace») для тех нажатий клавиш, которые не приводят к вводу текста.

Наиболее подходящим вариантом является обработка события PreviewTextInput (при нем происходит большая часть проверки) и использование события PreviewKeyDown для тех нажатий клавиш, которые не генерируют событие PreviewTextInput в текстовом поле (например, клавиша пробела). Ниже показано простое решение:

```
private void pnl_PreviewTextInput (object sender, TextComposi-
tionEventArgs e)
{
short val;
if (!(Int16.TryParse(e.Text, out val))
{
    // Запрет нажатий нечисловых клавиш,
    e.Handled = true;
}
}
private void pnlJPreviewKeyDown(object sender, KeyEventArgs e)
{
```

```
if (e.Key == Key.Space)
{
    e.Handled = true;
    }
}
```

Можно присоединить эти обработчики событий к одному текстовому полю или подключить их к контейнеру (например, к StackPanel, который содержит несколько текстовых полей для ввода чисел) для получения большей эффективности.

Такое поведение при обработке может показаться чрезвычайно неудобным. Одной из причин, по которой TextBox не может обеспечить лучшую обработку событий клавиатуры, является то, что WPF фокусируется на привязке данных – возможности, которая позволяет подключать элементы управления, такие как TextBox, к специальным объектам. При использовании этого подхода, проверка обычно выполняется ограничивающим объектом, ошибки подаются в виде исключения, а в случае неправильных данных генерируется сообщение об ошибке, которое появляется где-то в пользовательском интерфейсе.

5.5.5. Фокус ввода

В Windows пользователь может работать одновременно с одним элементом управления. Элемент управления, который в данный момент времени получает нажатия клавиши пользователем, находится вфокусеввода. Иногда такой элемент прорисовывается несколько иначе. Например, кнопка WPF приобретает синий оттенок, что свидетельствует о том, что она находится в фокусе ввода.

Чтобы элемент управления мог получать фокус, его свойству Focusable нужно присвоить значение true. По умолчанию оно является таковым для всех элементов управления.

Отметим, что свойство Focusable определяется как часть класса UIElement, а это означает, что остальные элементы, не являющиеся

элементами управления, тоже могут получать фокус. Обычно в классах, не определяющих элементы управления, свойство Focusable по умолчанию имеет значение false. Тем не менее, можно присвоить ему true. Например, если сделать это для контейнера StackPanel, то когда он будет получать фокус, по краям панели будет появляться пунктирная рамка.

Чтобы переместить фокус с одного элемента на другой, пользователь может щелкнуть кнопкой мыши или воспользоваться клавишей <Tab> и клавишами управления курсором. В предыдущих средах разработки программисты прилагали много усилий для того, чтобы клавиша <Tab> передавала фокус «по законам логики» (обычно слева направо, а затем вниз окна), и чтобы при первом отображении окна фокус передавался необходимому элементу управления. В WPF такая дополнительная работа требуется очень редко, поскольку WPF использует иерархическую компоновку элементов для реализации последовательности перехода с помощью клавиши табуляции. Фактически, при нажатии клавиши <Tab> происходит переход к первому потомку в текущем элементе или, если текущий элемент не имеет потомка, к следующему потомку, находящемуся на том же уровне. Например, если осуществляется переход с помощью клавиши табуляции в окне, в котором имеются два контейнера StackPanel, вначале будут пройдены все элементы управления в первом контейнере StackPanel, а затем все элементы управления во втором контейнере.

Если необходимо управлять последовательностью перехода с помощью клавиши табуляции, можно задать свойство TabIndex каждого элемента управления, чтобы определить его место в числовом порядке. Элемент управления, свойство TabIndex которого имеет значение 0, получает фокус первым, затем фокус получает элемент с большим значением этого свойства (например, 1, 2, 3 и т. д.). Если несколько элементов имеют одинаковые значения свойства TabIndex,

WPF выполняет автоматическую последовательность передачи фокуса, всоответствии *с* которой фокус получает ближайший элемент в последовательности.

Свойство TabIndex определяется в классе Control, вместе со свойством IsTabStop. Свойству IsTabStop можно присвоить значение false, чтобы исключить элемент управления из последовательности перехода с помощью клавиши табуляции. Различие между IsTabStop и Focusable заключается в том, что элемент управления, свойство IsTabStop которого имеет значение false, может получить фокус другим путем – либо программно (если в коде производится вызов метода Focus ()), либо по щелчку кнопкой мыши.

Элементы управления, являющиеся невидимыми или заблокированными (изображены серым цветом) обычно не включаются в последовательность перехода с помощью клавиши табуляции и не активизируются, независимо от настройки свойств TabIndex, IsTabStop и Focusable. Чтобы скрыть или заблокировать элемент управления, используются свойства Visibility и isEnabled, соответственно.

5.5.6. Получение состояния клавиши

Когда происходит нажатие клавиши, необходимо знать больше, чем просто то, какая именно клавиша была нажата. Кроме этого, нужно выяснить, какие еще клавиши были нажаты в это же время. Это означает, что необходимо изучить состояние остальных клавиш, особенно модификаторов вроде <Shift>, <Ctrl> и <Alt>.

События клавиш (PreviewKeyDown, KeyDown, PreviewKeyUp и KeyUp) способствуют получению этой информации. Во-первых, объект KeyEventArgs включает свойство KeyState, которое отражает свойство клавиши, сгенерировавшей событие. Еще есть свойство KeyboardDevice, которое предлагаетту же информацию для любого ключа на клавиатуре. Свойство KeyboardDevice предлагает экземпляр класса KeyboardDevice. Его свойства включают информацию о том,

какой элемент в данный момент имеет фокус (FocusedElement) и какие клавиши-модификаторы были нажаты в момент возникновения события (Modifiers). К клавишам-модификаторам относятся <Shift>, <Ctrl> и их состояние можно проверить с помощью следующего кода:

```
if ((e.KeyboardDevice.Modifiers & ModifierKeys.Control) == Mod-
ifierKeys.Control)
    {
     lblInfo.Text = "You held the Control key.";
}
```

KeyboardDevice предлагает несколько методов, которые перечислены в таблице 11. Каждому из них передается значение из перечисления Key.

Таблица 11

Методы KeyboardDevice

Имя	Описание
isKeyDown()	Сообщает о том, была ли нажата данная клавиша в момент возникновения события.
isKeyUp()	Сообщает о том, была ли отпущена данная клавиша в момент возникновения события.
isKeyToggled()	Сообщает о том, находилась ли данная клавиша во «включенном» состоянии в момент возникновения события. Это относится лишь к тем клавишам, которые могут включаться и выключаться: <CapsLock>, <ScrollLock> и <NumLock>.
GetKeyStates()	Возвращает одно или несколько значений из перечисления KeyStates, сообщающее о том, является ли данная клавиша нажатой, отпущенной, включена или выключена.

Когда используется свойство KeyEventArgs.KeyboardDevice, код получает состояние виртуальной клавиши. Это означает, что он получает состояние клавиатуры в момент возникновения события. Это может означать то же, что и текущее состояние клавиатуры. Например, представим, что произойдет, если пользователь вводит данные быстрее, нежели работает код. Каждый раз, когда будет возникать со-

бытие KeyPress, код будет иметь доступ к нажатию клавиши, сгенерировавшей событие, а не к символам, соответствующим нажатию. Как правило, именно такое поведение и оказывается необходимым.

Однако код не ограничен получением информации о клавише в событиях клавиатуры. Такжеможно получать состояние клавиатуры в любой момент времени. Для этой цели необходимо воспользоваться классом Keyboard, который очень похож на класс KeyboardDevice. за исключением того, что он создан из статических членов. Ниже показан пример, в котором класс Keyboard используется для проверки текущего состояния левой клавиши <Shift>:

```
if (Keyboard.IsKeyDown(Key.LeftShift))
{
   Ibllnfo.Text = "The left Shift is held down.";
}
```

5.5.7. Ввод с использованием мыши

События мыши решают несколько связанных задач. Самые главные события мыши позволяют определять действия в ответ на перемещение указателя мыши над элементом. Этими событиями являются MouseEnter (возникает, когда указатель мыши перемещается над элементом) и MouseLeave (происходит, когда указатель мыши покидает пределы элемента). Оба эти события являютсяпрямыми событиями (direct events), а это означает, что они не используют туннелирование или поднятие. Вместо этого они генерируются в одном элементе и продолжают свое существование только в нем. Такое поведение является оправданным и объясняется способами вложения элементов управления в окно WPF.

Например, если имеется панель StackPanel, в которой содержится кнопка, и указатель мыши наводится на эту кнопку, событие MouseEnter возникнет первым в элементе StackPanel (как только мышь войдет в пределы панели), а затем в кнопке (как только указа-

тель будет на нее наведен). Когда указатель мыши покинет пределы StackPanel, возникнет событие MouseLeave сначала в кнопке, а затем в StackPanel.

Также можно реагировать на два события, которые возникают при перемещении указателя мыши: PreviewMouseMove (туннельное событие) и MouseMove (событиеподнятия). Эти события предлагают код объекта MouseEventArgs. Этот объект включает свойства, которые могут сообщать о состоянии кнопок мыши в момент возникновения события, и метод GetPosition(), который сообщает координаты указателя мыши относительно выбираемого элемента. Ниже представлен пример, который отображает местонахождение указателя мыши относительно формы в независимых от устройства единицах:

```
private void MouseMoved(object sender, MouseEventArgs e)
{
  Point pt = e.GetPosition(this);
  lblInfo.Text = String.Format("You are at ({0},{1}) in window
coordinates", pt.X, pt.Y);
}
```

В данном случае координаты определяются, начиная с левого верхнего угла клиентской области (под строкой заголовка). На рис.46 виден результат выполнения этого кода.

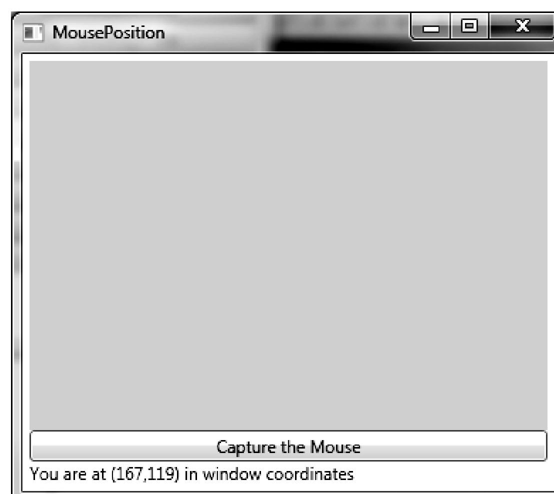

Рис. 44. Наблюдение за координатами мыши

Отметим, что координаты мыши в WPF не обязательно представлены целыми числами. Это объясняется тем, что приложение можетвыполняться в системе с разрешением, отличным от стандартного разрешения в 96 dpi. Как было сказано выше, WPF автоматически масштабирует свои единицы для компенсации, используя большее количество пикселей. Поскольку размер экранного пикселя больше не совпадает с размером в системе единиц WPF, физическое положение указателя мыши можно преобразовать в дробное число единиц WPF.

Класс UIElement содержит два полезных свойства, которые могут помочь в определении местоположения указателя мыши. С помощью свойства IsMouseOver можно определить, находится ли указатель мыши над элементом или одним из его потомков, а благодаря свойству isMouseDirectlyOver можно выяснить, располагается ли указатель мыши только над элементом, а не над его потомком. Как правило, в коде значения этих свойств не обрабатываются, а используются для создания триггеров стилей, которые автоматически изменяют элементы по мере перемещения указателя мыши над ними.

5.5.8. Щелчки кнопками мыши

Щелчки кнопками мыши подобны нажатиям клавиш на клавиатуре. Разница лишь в том, что события различаются для левой и правой кнопок. В таблице 12 перечислены события в порядке их возникновения. Помимо перечисленных, есть еще два события, которые реагируют на вращение колесика мыши: PreviewMouseWheel и MouseWheel.

Все события кнопок мыши имеют дело с объектом MouseButtonEventArgs. Класс MouseButtonEventArgs происходит от класса MouseEventArgs (а это означает, что он включает ту же информацию о координатах и состоянии кнопки) и добавляет несколько новых членов.

Таблица 12

События щелчков кнопками мыши в порядке возникновения

Имя	Тип маршрутизации	Описание
PreviewMouseLeftButtonDown PreviewMouseRightButtonDown	Туннелирование	Возникает при нажатии кнопки мыши.
MouseLeftButtonDown	Поднятие	Возникает при нажатии кнопки мыши.
PreviewMouseLeftButtonUp и PreviewMouseRightButtonUp	Туннелирование	Возникает при отпускании кнопки мыши.
MouseLeftButtonUp и MouseRightButtonUp	Поднятие	Возникает при отпускании кнопки мыши.

Менее важными свойствами являются MouseButton (сообщает о том, какая кнопка сгенерировала событие) и ButtonState (сообщает о том, в каком состоянии находилась кнопка в момент возникновения события: была нажата или отпущена). Интерес представляет свойство ClickCount, которое сообщает о том, сколько раз был произведен щелчок кнопкой, что позволит различать одиночные щелчки (в этом случае свойство будет иметь значение 1) и двойные щелчки (в этом случае свойство будет иметь значение 2).

Некоторые элементы добавляют высокоуровневые события мыши. Например, класс Control добавляет события PreviewMouseDoubleClick и MouseDoubleClick, которые замещают событие MouseLeftButtonUp. Точно так же, класс Button вызывает событие Click, которое могут сгенерировать клавиатура или мышь.

Как и события, возникающие при нажатии клавиши, события мыши предлагают информацию о том, в каком месте находился указатель мыши, и какой кнопкой был произведен щелчок в момент возникновения события. Для получения информации о текущей позиции указателя мыши и состоянии ее кнопок можно воспользоваться ста-

тическими членами класса Mouse, которые ничем не отличаются от статических членов класса MouseButtonEventArgs.

5.5.9. Захват мыши

Обычно каждый раз, когда элемент получает событие «down» кнопки мыши, через короткий промежуток времени он получает соответствующее событие «up» кнопки мыши. Однако так бывает не всегда. Например, если пользователь щелкает на элементе, удерживает нажатой кнопку мыши, а затем перемещает указатель мыши за пределы элемента, то элемент не получит событие отпускания кнопки мыши.

В некоторых ситуациях может понадобиться уведомление о событиях отпускания кнопки мыши, даже если они возникают после того, как указатель мыши покинул пределы элемента. Чтобы получать уведомления, нужнозахватить мышь, вызывая для этого метод Mouse.Capture() и передавая ему соответствующий элемент. С этого момента код будет получать события о нажатии и отпускании кнопок мыши до тех пор, пока снова не будет вызван метод Mouse.Capture() и не будет передана пустая (null) ссылка. Остальные элементы не получат события мыши до тех пор, пока мышь будет оставаться захваченной. Это означает, что пользователь не сможет щелкать кнопками мыши где-либо в окне, щелкать внутри текстовых полей и т. д. Захват мыши иногда используется для реализации функций перетаскивания и изменения размеров элементов.

При вызове метода Mouse.Capture() можно передавать необязательное значение в качестве второго параметра. Обычно при вызове метода Mouse.Capture() используется CaptureMode.Element, а это означает, что элемент будет всегда получать события мыши. Однако можно применить CaptureMode.SubTree для того, чтобы события мыши могли доходить до элемента, на котором был произведен щелчок кнопкой мыши, если этот элемент является потомком элемента, выполняющего захват. В этом есть смысл, если уже используется подня-

тие или туннелирование события для наблюдения за событиями мыши в дочерних элементах.

В некоторых случаях можно утратить захват мыши не по своей воле. Например, Windows может освободить мышь, если ей потребуется отобразить системное диалоговое окно. Это может случиться также в ситуации, если не освободить мышь после того, как возникнет событие, а пользователь переместит указатель, чтобы щелкнуть в окне в другом приложении. В любом случае можно реагировать на потерю захвата мыши, обрабатывая событие LostMouseCapture для данного элемента.

Пока мышь будет захвачена элементом, разработчик не сможет взаимодействовать с другими элементами. Захват мыши обычно используется в краткосрочных операциях, таких как перетаскивание.

5.5.10. Перетаскивание

Операции перетаскивания (способ изъятия информации из одного места в окне и переноса ее в другое место) сегодня не являются столь распространенными, как раньше. Программисты перешли на другие методы копирования информации, которые не требуют удержания нажатой кнопки мыши. Программы, которые поддерживают операцию перетаскивания, часто предлагают ее как быструю комбинацию для опытных пользователей, а не как стандартный способ работы.

Операция перетаскивания выполняется в три этапа, описанные ниже:

1. Пользователь щелкает на элементе (или выделяет некоторую область внутри него) и удерживает нажатой кнопку мыши. В этот момент начинается выполнение операции перетаскивания и сохраняется некоторая информация;
2. Пользователь наводит указатель мыши на другой элемент. Если этот элемент может принимать тип перетаскиваемого содержимого, указатель мыши принимает вид значка перетаскивания. В противном случае указатель мыши принимает вид перечеркнутого кружка;

3. Когда пользователь отпускает кнопку мыши, элемент получает информацию и принимает решение о дальнейшей ее судьбе. Эту операцию можно отменить, нажав клавишу <Esc> не отпуская кнопки мыши.

Можно потренироваться с операцией перетаскивания, добавив два объекта TextBox в окно, которое имеет код для поддержки операции перетаскивания. Если выбрать некоторый текст внутри текстового поля, то можно перетащить его в другое текстовое поле. Когда кнопка мыши будет отпущена, текст будет перемещен. Те же принципы распространяются и на приложения – например, можно перетащить фрагмент текста из документа Word в объект WPFTextBox или наоборот.

Иногда бывает необходимо выполнить перетаскивание между элементами, не обладающими такой возможностью. Например, нужно будет сделать так, чтобы пользователь мог перетаскивать содержимое из текстового окна на метку. Или создать пример, показанный на рис. 47, который дает пользователю возможность перетаскивать текст из объекта Label или TextBox в другую метку. В этой ситуации придется обрабатывать события перетаскивания.

Рис. 45. Форма для перетаскивания текста

Существуют две стороны операции перетаскивания: источник и цель. Чтобы создать источник перетаскивания, нужно вызвать метод DragDrop.DoDragDrop() в некоторой точке, чтобы начать операцию перетаскивания. В этот момент идентифицируется источник для операции перетаскивания, задается содержимое, которое нужно передавать, и указывается, какие эффекты будут разрешены при перетаскивании (копирование, перемещение и т. д.).

Обычно метод DoDragDrop() вызывается в ответ на событие MouseDown или PreviewMouseDown. Ниже показан пример, который начинает операцию перетаскивания при щелчке на метке. Для операции перетаскивания используется текстовое содержимое метки:

```
privatevoidlblSource_MouseDown(objectsender, MouseButtonEven-
tArgse)
{
     Label lbl = (Label)sender;
DragDrop.DoDragDrop(lbl, lbl.Content, DragDropEffects.Copy);
}
```

Элемент, который принимает данные, должен иметь в своем свойстве AllowDrop значение true. Кроме того, ему нужно обработать событие Drop, чтобы иметь возможность оперировать данными:

```
<Label Grid.Row="1" Grid.ColumnSpan="2" AllowDrop="True"
Drop="lblTarget_Drop">To this Label</Label>
```

Когда свойству AllowDrop присваивается значение true, элемент конфигурируется таким образом, чтобы разрешить любой тип информации. Если нужны большие возможности, можно обработать событие DragEnter. В этот момент можно проверить тип перетаскиваемых данных, а затем определить тип разрешенной операции. Следующий пример разрешает работать только с текстовым содержимым – если будет предпринята попытка перетащить что-то, что не может быть преобразовано в текст, операция перетаскивания не будет разрешена, а указатель мыши примет вид перечеркнутого кружка:

```
private void lblTarget_DragEnter(object sender, DragEventArgs
e)
{
if (e.Data.GetDataPresent(DataFormats.Text))
        e.Effects = DragDropEffects.Copy;
else
        e.Effects = DragDropEffects.None;
}
```

Наконец, когда операция будет завершена, можно извлечь данные и работать с ними. Следующий код принимает перемещенный текст и вставляет его в метку:

```
private void lblTarget_Drop(object sender, DragEventArgs e)
{
 ((Label)sender).Content = e.Data.GetData(DataFormats.Text);
}
```

Во время операции перетаскивания можно меняться объектами любых типов. Однако, несмотря на то, что этот простой способ прекрасно подходит для приложений, его применять не рекомендуется, если вам нужно связываться с другими приложениями. Если необходимо перетащить информацию в другое приложение, следует использовать базовый тип данных (например, строки, целые числа и т. п.) или объект, который мог бы реализовывать интерфейсы ISerializable или IDataObject (что позволит .NET передавать объект в поток байтов и заново создавать объект в другом домене приложения).

ЗАКЛЮЧЕНИЕ

Технология Windows Presentation Foundation, основы которой представлены в данном учебном пособии, является на сегодняшний день самой перспективной технологией создания пользовательских интерфейсов программных продуктов. Основным преимуществом технологии является возможность разделения труда между разработчиком интерфейсов и разработчиком программной логики. Очевидно, что подобное разделение позволяет ускорить разработку программного обеспечения и повысить его качество.

Рассматриваемая в учебном пособии технология не исчерпывается контейнерами компоновки и обработкой маршрутизируемых событий, рассмотренных в пособии. Также в рамках Windows Presentation Foundation определены ресурсы и стили для оформления элементов пользовательского интерфейса, механизмы привязки данных к интерфейсным элементам и шаблоны, позволяющие разработчику переопределить внешний вид всех элементов управления, включая стандартные, причем имеется возможность учитывать привязанные данные. Также в рамках рассматриваемой технологии имеются возможности применения к элементам управления трансформаций и анимаций, а также возможность использования трехмерной графики. Подробнее о механизмах, не вошедших в данное учебное пособие, можно прочитать в [1] и [2], а также по следующим адресам:

- http://msdn.microsoft.com/ru-ru/library/ms754130.aspx;
- http://msdn.microsoft.com/ru-ru/netframework/aa663326.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Border, 112, 113, 122
Canvas, 45, 76, 77, 78, 79, 161
Content, 36, 92, 94, 106, 149, 150, 162
DirectX, 8, 10, 11, 15, 157
DockPanel, 45, 53, 55, 56, 57, 64, 65, 75, 87, 89, 159, 160, 161
Expander, 92, 98, 104, 107, 108, 109, 110, 111, 155, 162
Grid, 22, 23, 25, 27, 28, 29, 31, 32, 35, 43, 45, 53, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 79, 82, 83, 84, 85, 86, 87, 89, 90, 97, 98, 99, 109, 113, 114, 115, 121, 123, 124, 128, 150, 154, 159, 160, 161
GridSplitter, 67, 68, 69, 70, 71, 73, 161
GroupBox, 92, 98, 104, 105, 162
InkCanvas, 45, 78, 79, 80, 81, 82
Panel, 19, 44, 79, 90, 92
ScrollViewer, 44, 91, 98, 99, 100, 101, 102, 103, 111, 162
StackPanel, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 65, 67, 75, 86, 87, 89, 90, 93, 103, 104, 105, 106, 108, 109, 112, 113, 119, 120, 121, 124, 126, 127, 128, 139, 140, 143, 154, 159, 160, 161
TabItem, 92, 98, 104, 105, 106, 107, 162
UniformGrid, 45, 75, 155
Viewbox, 112, 113, 114, 115
WarpPanel, 45, 88
Windows Forms, 7, 8, 13, 14, 15, 20, 41, 42, 58, 77
Windows Presentation Foundation, 7
WindowsForms, 157
WrapPanel, 45, 53, 54, 56, 87, 109, 154, 160
XAML, 13, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 38, 39, 59, 63, 78, 117, 118, 123, 125, 127, 131, 154, 157, 158
Декораторы, 111, 155, 162
Захват мыши, 146, 147
Маршрутизация событий, 117, 119, 155
Перетаскивание, 147
Прикрепленные свойства, 34, 160
Расширение разметки, 33
Свойства зависимостей, 115, 116
Туннельные события, 120, 128, 129, 163
Туннельныесобытия, 128
Элементы управления содержимым, 89, 90, 95, 104, 155, 161

ГЛОССАРИЙ

Декораторы – элементы управления содержимым, предназначенные для украшения области вокруг элемента управления.

Маршрутизируемые события (routedevents) – события, основанные на новой концепциимаршрутизации событий. маршрутизация позволяет событию возникать в одном элементе, а генерироваться – в другом. Маршрутизация событий предлагает большую гибкость для написания лаконичного кода, который сможет обрабатывать события в более удобном для этого месте.

Компоновка – процесс разработки пользовательского интерфейса, который состоит в распределении пространства формы между контейнерами компоновки и наполнения контейнеров элементами управления. Основная трудность компоновки заключается в том, что дизайн формы должен автоматически адаптироваться к изменениям ее размера.

Контейнер компоновки – элемент, который содержит в себе один или более дочерних элементов и упорядочивает их в соответствии с определенными правилами компоновки.

Содержимое – все, что находится внутри элемента управления, включая текст, графику, мультимедийную информацию, а также другие элементы управления.

Expander – элемент управления содержимым, который упаковывает область содержимого, которую пользователь может показывать или скрывать, щелкая на небольшой кнопке со стрелкой. Эта технология используется часто в оперативных справочных системах, а также на Web-страницах, чтобы они могли включать большие объемы содержимого, не перегружая пользователей информацией, которую им не хочется видеть.

GPU – узел обработки графики, процессор видеокарты компьютера.

Grid – контейнер компоновки, размещающий свои дочерние элементы в ячейках невидимой сетки. Обладает наибольшими возможностями среди всех контейнеров компоновки.

StackPanel – контейнер компоновки, располагающий свои дочерние элементы в столбец или строку.

UniformGrid – контейнер компоновки, размещающий дочерние элементы по сетке с одинаковыми клетками. Размер клеток рассчитывается автоматически.

XAML (сокращение от Extensible Application Markup Language – расширяемый язык разметки приложений) – язык декларативного описания пользовательских интерфейсов в технологии Windows Presentation Foundation.

Windows Presentation Foundation – новая технология создания пользовательских интерфейсов, одной из основных возможностей которой является декларативное описание пользовательского интерфейса. Благодаря этому создание пользовательского интерфейса приложения может быть разделено между программистом и дизайнером, что позволяет сократить сроки разработки программного обеспечения и повысить его качество.

WrapPanel – контейнер компоновки, располагающий свои дочерние элементы в столбец или строку. В отличие от StackPanel, данный контейнер при нехватке места для размещения элементов начинает новую строку или столбец

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Мак-Дональд, М. WPF: Windows Presentation Foundation в .NET 4.0 с примерами на C# 2010 для профессионалов / М. Мак-Дональд. – М. : Вильямс, 2011. – 1024 с.
2. Nathan, Adam WindowsPresentationFoundationUnleashed / A. Nathan, D. Lehenbauer. – Sams Publishing, 2007. – 656 с.

ПРИЛОЖЕНИЕ

Примеры тестовых заданий

1. Основнымпреимуществом Windows Presentation Foundation перед Windows Forms является:
   - Декларативное описание пользовательского интерфейса и возможность разделения работы программиста и дизайнера;
   - Возможность повторного использования дизайна пользовательского интерфейса;
   - Независимость от разрешения устройства отображения
   - Использование DirectX для отображения пользовательского интерфейса.
2. Укажите количество слоев архитектуры Windows Presentation Foundation:
   - 2;
   - 3;
   - 4;
   - 1.
3. Укажите название языка разметки, используемого в Windows Presentation Foundation:
   - XML;
   - HTML;
   - XAML;
   - SOAP.
4. Основным инструментом разработчика является Visual Studio. Назовите основной рабочий инструмент дизайнера пользовательского интерфейса:
   - Microsoft Expression Blend;
   - Adobe Photoshop;

- Microsoft Visio;
- GIMP.

5. Определите существующие подмножества XAML:
   - HTML XAML;
   - WPF XAML;
   - WWF XAML;
   - XPS XAML.
6. Укажите способ присваивания значений свойствам класса:
   - Непосредственное присваивание;
   - Присваивание через атрибуты;
   - Присваивание в конструкторе класса;
   - Создание наследника класса.
7. Укажите возможное количество элементов верхнего уровня в документе XAML:
   - 1;
   - 2;
   - 3;
   - Ни одного.
8. Какой атрибут следует использовать при необходимости включить в текст несколько пробелов подряд:
   - xml: space= "preserve";
   - xml: space= "complete";
   - xml: space= "save";
   - xml: space= "ignore".
9. Основной трудностью при компоновке элементов пользовательского интерфейса является:
   - размещение элементов;
   - определение необходимых элементов;
   - создание привлекательной компоновки;
   - способность адаптироваться к различным размерам окна.

10. Компоновка элементов в WPFоснована на:
   - координатах;
   - контейнерах;
   - сетках;
   - на порядке следования элементов.
11. Является ли необходимым задание явных координат и размеров элементов в WPF?
   - Нет;
   - Да;
   - Да, в некоторых случаях;
   - Нет, в особых случаях.
12. Укажите количество стадий процесса компоновки:
   - 1;
   - 2;
   - 3;
   - 4.
13. Назовите контейнер, размещающий свои дочерние элементы в столбец или строку:
   - StackPanel;
   - DockPanel;
   - Grid;
   - VirtualizingStackPanel.
14. Каким образом определяются размеры элемента управления при компоновке?
   - Фиксированными размерами;
   - По размеру содержимого автоматически;
   - По размеру содержимого вручную;
   - Произвольным образом.
15. Каким образом можно явно задать размеры элементы управления?

- Явно заданными числами;
- Максимальным и минимальным размерами;
- Произвольным образом;
- Явно задать размеры элемента управления невозможно.

16. Контейнер компоновки, позволяющий создавать новые строки или столбцы элементов, называется:
    - StackPanel;
    - WrapPanel;
    - DockPanel;
    - Grid.
17. Контейнер компоновки, позволяющий располагать элементы управления вдоль своих границ, называется:
    - StackPanel;
    - WrapPanel;
    - DockPanel;
    - Grid.
18. Свойство LastChildFill контейнера DockPanel используется для:
    - Заполнения остатка окна прочим содержимым;
    - Управления выделением памяти;
    - Организацией отображения элементов интерфейса;
    - Свойство не используется.
19. Контейнер, располагающий элементы управления в ячейках невидимой сетки, называется:
    - StackPanel;
    - WrapPanel;
    - DockPanel;
    - Grid.
20. Для размещения элементов в сетке контейнера Gridнеобходимо использовать:
    - Прикрепленные свойстваRow и Column;

- Свойства элемента Row и Column;
- Средства среды разработки;
- Создание контейнера Gridпосле размещения элементов.

21. Укажите стратегию измерения размеров, которую не использует контейнер Grid:
    - Абсолютные размеры;
    - Автоматические размеры;
    - Пропорциональные размеры;
    - Мировые размеры.
22. Свойства RowSpan и ColumnSpan используются для:
    - Объединения строк и колонок;
    - Разбиения строк и колонок;
    - Заполнения строк и колонок содержимым;
    - Удаления строк и колонок.
23. Полосы разделителей в WPF представлены классом:
    - GridSplitter;
    - RowSplitter;
    - ColumnSplitter;
    - Splitter.
24. Контейнер компоновки, позволяющий размещать элементы, используя точные координаты, называется:
    - Canvas;
    - DockPanel;
    - StackPanel;
    - Grid.
25. Специализированный тип элементов управления, которые могут хранить и отображать какую-то порцию содержимого, называется:
    - Элементы управления содержимым;
    - Элементы управления;

- Отображаемые элементы;
- Элементы управления поведением.

26. Содержимое элемента управления содержимым определяется свойством:
    - Content;
    - ContentControl;
    - ContentPlaceholder;
    - Frame.
27. В WPF поддержка прокрутки осуществляется при помощи:
    - ScrollViewer;
    - GroupBox;
    - TabItem;
    - Expander.
28. Укажите класс, не являющийся наследником класса ContentControl:
    - GroupBox;
    - TabItem;
    - Expander;
    - ScrollViewer.
29. Элемент управления, упаковывающий область содержимого, которую пользователь может показывать или скрывать, щелкая на небольшой кнопке со стрелкой, называется:
    - GroupBox;
    - TabItem;
    - Expander;
    - ScrollViewer.
30. Контейнеры, служащие для украшения области вокруг объекта, называются:
    - Декораторы;
    - Контейнеры компоновки;

- Контейнеры поведения;
- Прокрутчики.

31. Декоратор ViewBox предназначен для:
    - Масштабирования содержимого;
    - Масштабирования элемента управления;
    - Для управления содержимым;
    - Для маскировки элемента управления.
32. Для событий в WPF используется концепция:
    - Маршрутизации событий;
    - Подавления событий;
    - Возникновения событий;
    - Стандартная концепция.
33. Укажите вид событий, который не является маршрутизируемым:
    - Прямые события;
    - Поднимающиеся события;
    - Туннельные события;
    - Управляемые события.
34. Упорядочите события клавиатуры в порядке возникновения:
    - PreviewKeyDown;
    - KeyDown;
    - PreviewTextInput;
    - TextInput;
    - PreviewKeyUp;
    - KeyUp.
35. Упорядочите события щелчков кнопками мыши в порядке возникновения:
    - PreviewMouseLeftButtonDown;
    - PreviewMouseRightButtonDown;
    - MouseLeftButtonDown;
    - PreviewMouseLeftButtonUp;
    - PreviewMouseRightButtonUp;
    - MouseLeftButtonUp, MouseRightButtonUp.

Учебное издание

ШАМШЕВ Анатолий Борисович

ОСНОВЫ ПРОЕКТИРОВАНИЯ ИНТЕРФЕЙСОВ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ WINDOWS PRESENTATION FOUUNDATION

Учебное пособие

Редактор М. В. Штаева

ЛР №020640 от 22.10.97.
Подписано в печать 15.03.2012. Формат 60×84/16.
Усл. печ. л. 9,53. Тираж 125 экз. Заказ 267.

Ульяновский государственный технический университет
432027, г. Ульяновск, Сев. Венец, д. 32.

Типография УлГТУ, 432027, г. Ульяновск, Сев. Венец, д. 32.