ГЛАВА 4

Компоновка

Половина всех усилий при проектировании пользовательского интерфейса уходит на организацию содержимого, чтобы она была привлекательной, практичной и гибкой. Но реальный вызов состоит в том, чтобы убедиться в том, что ваша компоновка элементов интерфейса сможет успешно адаптироваться к разным размерам окна.

В WPF вы разделяете компоновку, используя разные *контейнеры*. Каждый контейнер имеет свою собственную логику компоновки — некоторые складывают элементы в стопку, другие распределяют их по ячейкам сетки и т.д. Если вы программировали с применением Windows Forms, то будете удивлены, что компоновка на основе координат в WPF не рекомендована к использованию. Вместо этого упор делается на создание более гибких компоновок, которые могут адаптироваться к изменяющемуся содержимому, разным языкам и широкому разнообразию размеров окон. Для большинства разработчиков переход к WPF с его новой системой компоновки становится большим сюрпризом — и первой реальной сложностью.

В этой главе вы увидите, как работает модель компоновки WPF, и начнете использовать базовые контейнеры компоновки. Также вы рассмотрите несколько примеров распространенной компоновки — от базового диалогового окна к разделенному окну изменяемого размера, — чтобы изучить основы компоновки WPF.

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 Windows Forms представил весьма примитивную систему компоновки. Элементы управления были фиксированы на месте по жестко закодированным координатам. Единственным удобством были *привязка* (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 (с равномерным разделением дополнительного пространства между двумя областями). Они также не слишком помогали в случае высокодинамичного содержимого, например, когда нужно было дать возможность метке расширяться, чтобы вместить больше текста, что приводило к перекрытию соседних элементов управления.

В .NET 2.0 каркас Windows Forms заполнил пробел, благодаря двум новым контейнерам компоновки: `FlowLayoutPanel` и `TableLayoutPanel`. Используя эти элементы

управления, вы можете создавать более изощренные интерфейсы в стиле Web. Оба контейнера компоновки позволяли содержащимся в них элементам управления увеличиваться, расталкивая соседние элементы. Это облегчило задачу создания динамического содержимого, создания модульных интерфейсов и локализации вашего приложения. Однако панели компоновки выглядели дополнением к основной системе компоновки Windows Forms, использовавшей фиксированные координаты. Панели компоновки были элегантным решением, но все-таки несколько чужеродным.

WPF предлагает новую систему компоновки, которая была навеяна опытом разработки в Windows Forms. Эта система возвращает модель .NET 2.0 (координатную компоновку с необязательными потоковыми панелями компоновки), 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, подгоняют себя при изменении содержимого и легко обрабатывают перевод на другие языки. Однако прежде чем вы воспользуетесь преимуществом этих изменений, вам следует перестроить ваш образ мышления относительно компоновки.

Философия компоновки WPF

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

На заметку! Это ограничение обусловлено тем фактом, что класс `Window` наследуется от `ContentControl`, который вы изучите более подробно в главе 5.

В WPF компоновка определяется используемым контейнером. Хотя есть несколько контейнеров, среди которых можно выбирать, “идеальное” окно WPF следует описанным ниже ключевым принципам.

- *Элементы (такие как элементы управления) не должны иметь явно установленных размеров.* Вместо этого они растут, чтобы вместить их содержимое. Например, кнопка увеличивается, когда вы добавляете в нее текст. Вы можете 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 (необязательно) другой информации, специфичной для контейнера компоновки. Если вы хотите добавить пробел между элементами, то используете для этого свойство `Margin`.

На заметку! Жестко закодированные размеры позиции — зло, потому что они ограничивают ваши возможности по локализации интерфейса и значительно затрудняют работу с динамическим содержимым.

- *Контейнеры компоновки “разделяют” доступное пространство между своими дочерними элементами.* Они пытаются предоставить каждому элементу его предпочтительный размер (на основе его содержимого), если позволяет свободное пространство. Они могут также выделять дополнительное пространство одному или более дочерним элементам.
- *Контейнеры компоновки могут быть вложенными.* Типичный пользовательский интерфейс начинается с `Grid` — наиболее развитого контейнера, и содержит дру-

гие контейнеры компоновки, 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. Другими словами, если вы последуете этим руководствам при построении WPF-приложения, то получите лучший, более гибкий пользовательский интерфейс. Если же вы нарушаете эти правила, то получите пользовательский интерфейс, который не будет хорошо подходить для WPF и его будет значительно сложнее сопровождать.

Процесс компоновки

Компоновка WPF происходит за две стадии: стадия *измерения* и стадия *расстановки*. 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 позиции.

Конечно, элемент не может всегда иметь свой предпочтительный размер — иногда контейнер недостаточно велик, чтобы обеспечить его. В этом случае контейнер должен усекать такой элемент для того, чтобы вместить его в видимую область. Как вы убедитесь, часто можно избежать такой ситуации, устанавливая минимальный размер окна.

На заметку! Контейнеры компоновки не поддерживают прокрутку. Вместо этого прокрутка обеспечивается специализированным элементом управления содержимым — `ScrollViewer`, — который может быть использован почти где угодно. В главе 5 вы узнаете больше о `ScrollViewer`.

Контейнеры компоновки

Все контейнеры компоновки WPF являются панелями, которые унаследованы от абстрактного класса `System.Windows.Controls.Panel` (рис. 4.1). Класс `Panel` добавляет небольшой набор членов, включая три общедоступных свойства, описанные в табл. 4.1.

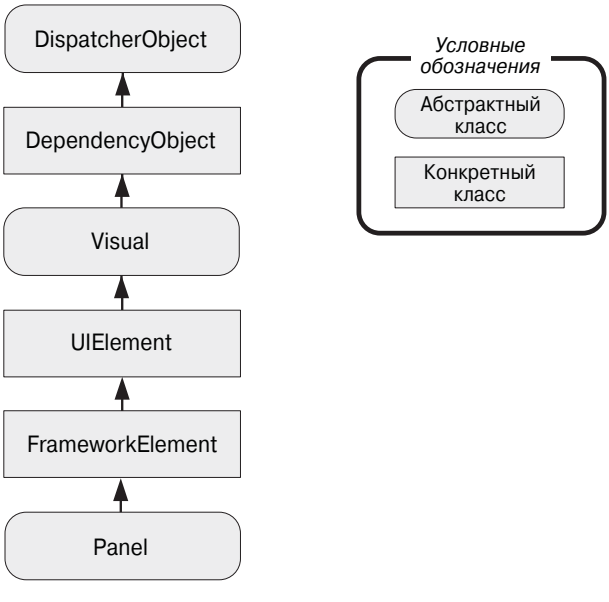

Рис. 4.1. Иерархия класса `Panel`

Таблица 4.1. Общедоступные свойства класса `Panel`

Имя	Описание
`Background`	Кисть, используемая для рисования фона панели. Вы должны устанавливать это свойство в отличное от `null` значение, если хотите принимать события мыши. (Если вы хотите принимать события мыши, но не желаете отображать сплошной фон, просто установите прозрачный цвет фона — `Transparent`.) Из главы 7 вы узнаете больше о базовых кистях (о более развитых кистях читайте в главе 13).
`Children`	Коллекция элементов, находящихся в панели. Это первый уровень элементов — другими словами, это элементы, которые сами могут содержать в себе другие элементы.
`IsItemsHost`	Булевское значение, равное `true`, если панель используется для показа элементов, ассоциированных с `ItemsControl` (такие как узлы из `TreeView` или элементы списка `ListBox`). Большую часть времени вы даже не будете знать о том, что элемент-список используется "за кулисами" панелью для управления компоновкой элементов. Однако эта деталь становится более важной, если вы хотите создать специальный список, который будет располагать свои дочерние элементы другим способом (например, `ListBox`, отображающий графические изображения). Вы используете эту технику в главе 17.

На заметку! Класс `Panel` также имеет некоторый внутренний механизм, который вы можете использовать, если хотите создать собственный контейнер компоновки. Но важнее то, что вы можете переопределить методы `MeasureOverride()` и `ArrangeOverride()`, унаследованные от `FrameworkElement`, для изменения способа обработки панелью стадий измерения и расстановки при организации дочерних элементов. Вы узнаете, как создать специальную панель в главе 24.

Сам по себе базовый класс `Panel` — это не что иное, как начальная точка для построения других более специализированных классов. WPF предлагает ряд производных от `Panel` классов, которые вы можете использовать для организации компоновки. Самые основные перечислены в табл. 4.2. Как и все элементы управления WPF, а также большинство визуальных элементов, эти классы находятся в пространстве имен `System.Windows.Controls`.

Таблица 4.2. Основные панели компоновки

Имя	Описание
`StackPanel`	Размещает элементы в горизонтальный или вертикальный стек. Этот контейнер компоновки обычно используется в небольших секциях крупного более сложного окна.
`WrapPanel`	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.

Окончание табл. 4.2

Имя	Описание
`UniformGrid`	Помещает элементы в невидимую таблицу, устанавливая одинаковый размер для всех ячеек. Данный контейнер компоновки используется нечасто.
`Canvas`	Позволяет элементам позиционироваться абсолютно — по фиксированным координатам. Этот контейнер компоновки более всего похож на традиционный компоновщик Windows Forms, но не предусматривает средств привязки и стыковки. В результате это неподходящий выбор для окон переменного размера, если только вы не собираетесь взвалить на свои плечи значительный объем работы.

Наряду с этими центральными контейнерами есть еще несколько более специализированных панелей, которые вы встретите 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. (Например, в зависимости от выбранного режима, `InkCanvas` поддерживает рисование с указателем для выбора экранных элементов. Хотя это не очень удобно, но вы можете использовать `InkCanvas` на обычном компьютере с мышью.)

Простая компоновка с помощью `StackPanel`

`StackPanel` — один из простейших контейнеров компоновки. Он просто укладывает свои дочерние элементы в одну строку или колонку.

Например, рассмотрим следующее окно, которое содержит стек из трех кнопок:

```
<Window x:Class="Layout.SimpleStack"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Layout" Height="223" Width="354"
    >
  <StackPanel>
    <Label>A Button Stack</Label>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
  </StackPanel>
</Window>
```

На рис. 4.2 показанное полученное в результате окно.

Использование `StackPanel` в Visual Studio

Этот пример сравнительно просто создать, используя дизайнер Visual Studio. Начните с удаления корневого элемента `Grid` (если он есть). Затем перетащите `StackPanel` в окно. После этого перетащите другие элементы (метку и четыре кнопки) в окно, в желаемом порядке сверху вниз.

Если вы хотите реорганизовать элементы в `StackPanel`, вы не можете просто перетаскивать их. Вместо этого выполните щелчок правой кнопкой мыши на нужном элементе и выберите в контекстном меню команду Order (Упорядочить). Опции упорядочивания соответствует поряд-

ку элементов в разметке, с первым элементом, занимающим последнюю позицию, и последним элементом, занимающим первую позицию. Таким образом, вы можете перемещать элемент вниз панели `StackPanel` (используя Bring to Front (На передний план)), вверх (используя Send to Back (На задний план)) либо на одну позицию вверх или вниз (используя Bring Forward (Вперед) и Send Backward (Назад)).

При создании пользовательского интерфейса в Visual Studio вы должны учитывать некоторые нюансы. Когда вы перетаскиваете элементы из панели инструментов в окно, Visual Studio добавляет некоторые детали в вашу разметку. Среда Visual Studio автоматически присваивает имя каждому новому элементу управления (что безвредно, но излишне). Также добавляются жестко закодированные значения `Width` и `Height`, что ограничивает намного больше.

Как уже говорилось ранее, явные размеры ограничивают гибкость вашего пользовательского интерфейса. Во многих случаях лучше позволить элементам управления самостоятельно устанавливать свои размеры, подгоняя их к контейнеру. В данном примере фиксированные размеры представляют собой оправданный подход, чтобы установить для всех кнопок согласованную ширину. Однако более удачное решение состояло бы в том, чтобы позволить самой большой кнопке устанавливать свой размер самостоятельно, вмещая свое содержимое, а все остальные кнопки — растянуть до размера большой, чтобы они соответствовали друг другу. (Такой дизайн, требующий применения `Grid`, описан далее в этой главе.) Но независимо от того, какой подход вы используете с кнопкой, почти наверняка вы захотите избавиться от жестко закодированных величин `Width` и `Height` для `StackPanel`, чтобы она могла растягиваться и сжиматься, заполняя доступное пространство окна.

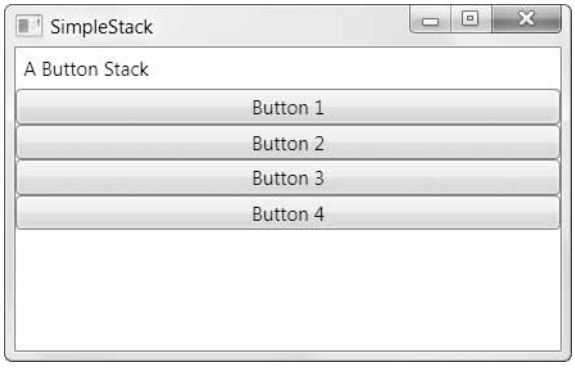

Рис. 4.2. `StackPanel` в действии

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 вы расширите окно, `StackPanel` также расширится, и кнопки растянутся, чтобы заполнить ее.

`StackPanel` может также использоваться для упорядочивания элементов в горизонтальном направлении посредством установки свойства `Orientation`:

```
<StackPanel Orientation="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. 4.3.

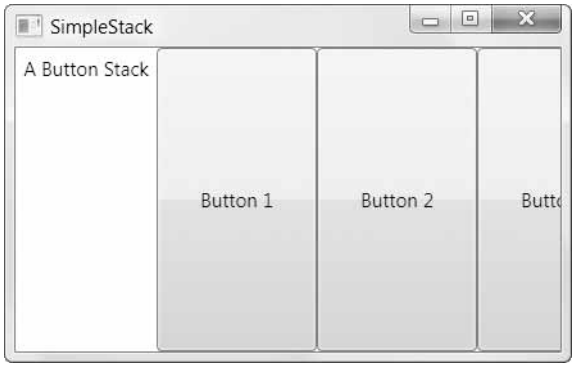

Рис. 4.3. `StackPanel` с горизонтальной ориентацией

Ясно, что это не обеспечивает достаточной гибкости, необходимой реальному приложению. К счастью, вы можете тонко настраивать способ работы `StackPanel` и других контейнеров компоновки посредством свойств компоновки, как описано ниже.

Свойства компоновки

Хотя компоновка определяется контейнером, дочерние элементы тоже могут сказать свое слово. Фактически панели компоновки взаимодействуют со своими дочерними элементами через небольшой набор свойств компоновки, перечисленных в табл. 4.3.

Таблица 4.3. Свойства компоновки

Наименование	Описание
`HorizontalAlignment`	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по горизонтали. Вы можете выбрать `Center`, `Left`, `Right` или `Stretch`.
`VerticalAlignment`	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по вертикали. Вы можете выбрать `Center`, `Top`, `Bottom` или `Stretch`.
`Margin`	Добавляет немного места вокруг элемента. Свойство `Margin` — это экземпляр структуры `System.Windows.Thickness`, с отдельными компонентами для верхней, нижней, левой и правой граней.
`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 (виджетами), которые вы можете использовать в окне WPF.

На заметку! Как вам известно из главы 2, различные контейнеры компоновки могут представлять *прикрепленные свойства* к их дочерним элементам. Например, все дочерние элементы объекта `Grid` получают свойства `Row` и `Column`, которые позволяют им выбирать ячейку, в которой они должны разместиться. Прикрепленные свойства позволяют устанавливать информацию, специфичную для определенного контейнера компоновки. Однако свойства компоновки из таблицы носят достаточно общий характер, чтобы применяться ко многим панелям компоновки. Таким образом, эти свойства определены как часть базового класса `FrameworkElement`.

Этот список свойств замечателен тем, чего он *не* содержит. Если вы ищете знакомые свойства позиционирования, такие как `Top`, `Right` и `Location`, вы не найдете их там. Это потому, что большинство контейнеров компоновки (кроме `Canvas`) используют автоматическую компоновку и не дают вам возможности явного позиционирования элементов.

Выравнивание

Чтобы понять, как работают эти свойства, еще раз взглянем на простую `StackPanel`, показанную на рис. 4.2. 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. Вот почему каждая кнопка целиком занимает ширину колонки. Однако вы можете изменить эти детали:

```
<StackPanel>
  <Label HorizontalAlignment="Center">A Button Stack</Label>
  <Button HorizontalAlignment="Left">Button 1</Button>
  <Button HorizontalAlignment="Right">Button 2</Button>
  <Button>Button 3</Button>
  <Button>Button 4</Button>
</StackPanel>
```

На рис. 4.4 показан результат. Первые две кнопки получают минимальные размеры и выравниваются соответственно, в то время как две нижние кнопки растянуты на всю `StackPanel`. Если вы измените размер окна, то увидите, что метка остается в середине, а первые две кнопки будут прижаты каждая к своей стороне.

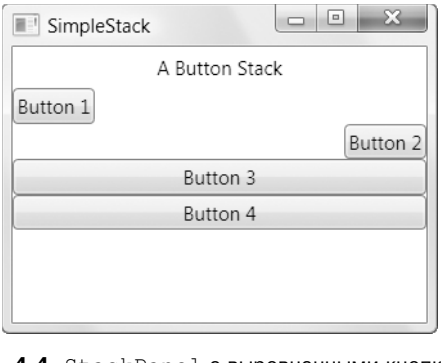

Рис. 4.4. `StackPanel` с выровненными кнопками

На заметку! StackPanel также имеет собственные свойства HorizontalAlignment и VerticalAlignment. По умолчанию оба они установлены в Stretch, и потому StackPanel заполняет свой контейнер полностью. В данном примере это значит, что StackPanel заполняет окно. Если вы используете другие установки, максимальный размер StackPanel будет определяться самым широким элементом управления, содержащимся в нем.

Поля

В текущей форме примера StackPanel присутствует очевидная проблема. 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", вы можете устанавливать поля вокруг элементов управления.

При установке полей вы можете установить одинаковую ширину для всех сторон, как здесь:

```
<Button Margin="5">Button 3</Button>
```

Альтернативно вы можете установить разные поля для каждой стороны элемента управления в порядке *левое*, *верхнее*, *правое*, *нижнее*:

```
<Button Margin="5,10,5,10">Button 3</Button>
```

В коде поля устанавливаются в структуре Thickness:

```
cmd.Margin = new Thickness(5);
```

Определение правильных полей вокруг элементов управления — отчасти искусство, потому что вы должны учитывать, каким образом установки полей соседних элементов управления влияют друг на друга. Например, если у вас есть две кнопки, сложенные одна на другую, и самая верхняя кнопка имеет нижнее поле размером 5, а самая нижняя кнопка имеет верхнее поле размером 5, то у вас получится пространство в 10 единиц между двумя кнопками.

В идеале вы сможете 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 показано ниже:

```
<StackPanel Margin="3">
  <Label Margin="3" HorizontalAlignment="Center">
   A Button Stack</Label>
  <Button Margin="3" HorizontalAlignment="Left">Button 1</Button>
  <Button Margin="3" HorizontalAlignment="Right">Button 2</Button>
  <Button Margin="3">Button 3</Button>
  <Button Margin="3">Button 4</Button>
</StackPanel>
```

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). На рис. 4.5 показано наиболее приемлемое окно, а на рис. 4.6 — как его изменяют установки полей.

Минимальный, максимальный и явный размеры

И, наконец, каждый элемент включает свойства Height и Width, которые позволяют вам установить явный размер. Однако предпринимать такой шаг — не слишком хорошая идея.

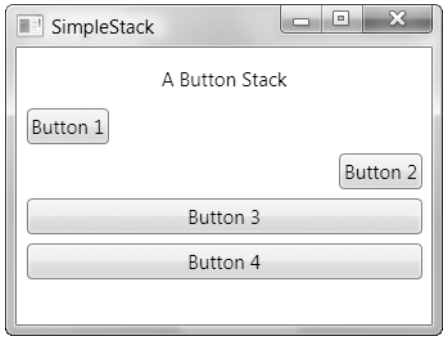

Рис. 4.5. Добавление полей между элементами

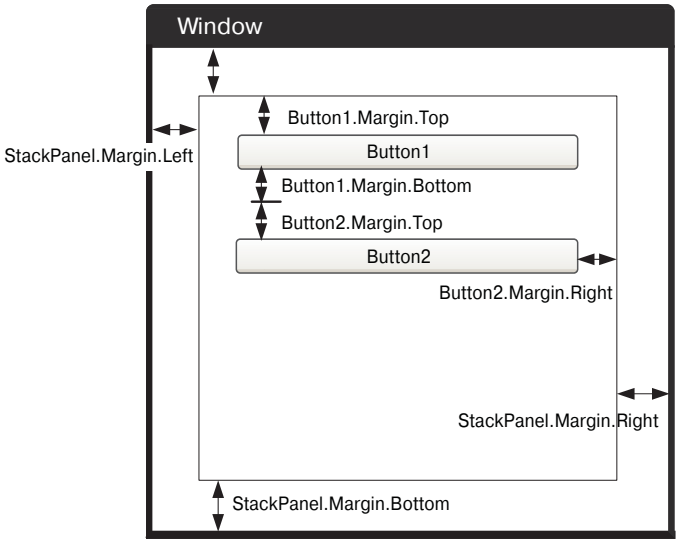

Рис. 4.6. Как комбинируются поля

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

Совет. Подумайте дважды, прежде чем устанавливать явные размеры в WPF. В хорошо спроектированной компоновке в этом не должно быть необходимости. Если вы добавляете информацию о размерах, то рискуете создать хрупкую компоновку, которая не может адаптироваться к изменениям (таким как разные языки и размеры окон) и усекает ваше содержимое.

Например, вы можете решить, что кнопки в вашей `StackPanel` должны растягиваться для ее заполнения, но быть не более 200 единиц и не меньше 100 единиц в ширину. (По умолчанию, кнопки начинаются с минимальной ширины в 75 единиц.) Вот какая разметка вам для этого понадобится:

```
<StackPanel Margin="3">
  <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>
</StackPanel>
```

Совет. Здесь вы можете спросить: а нет ли более простого способа установки свойств, стандартизованных для нескольких элементов, таких как поля кнопок в этом примере? Ответом могут служить *стили* — средство, позволяющее вам повторно использовать установки свойств и даже применять их автоматически. Подробнее о стилях вы узнаете из главы 12.

Когда `StackPanel` изменяет размеры кнопки, он учитывает несколько единиц информации.

- *Минимальный размер*. Каждая кнопка всегда будет не меньше минимального размера.
- *Максимальный размер*. Каждая кнопка всегда будет меньше максимального размера (если только вы не установите неправильно максимальный размер меньше минимального).
- *Содержимое*. Если содержимое внутри кнопки требует большей ширины, то `StackPanel` попытается увеличить кнопку. (Вы можете определить размер, который нужен кнопке, проверив свойство `DesiredSize`, которое вернет минимальную ширину или ширину содержимого — в зависимости от того, что больше.)
- *Размер контейнера*. Если минимальная ширина больше, чем ширина `StackPanel`, то часть кнопки будет усечена. В противном случае кнопке не позволено будет расти шире, чем позволит `StackPanel`, даже несмотря на то, что она не сможет вместить весь текст на своей поверхности.
- *Горизонтальное выравнивание*. Поскольку кнопка использует `HorizontalAlignment`, равный `Stretch` (по умолчанию), `StackPanel` попытается увеличить кнопку, чтобы она заполнила полную ширину `StackPanel`.

Сложность понимания этого процесса заключается в том, что минимальный и максимальный размер устанавливает абсолютные пределы. Без этих пределов `StackPanel` пытается обеспечить желаемый размер кнопки (чтобы вместить ее содержимое) и установки выравнивания.

Рис. 4.7 проливает некоторый свет на то, как это работает в `StackPanel`.

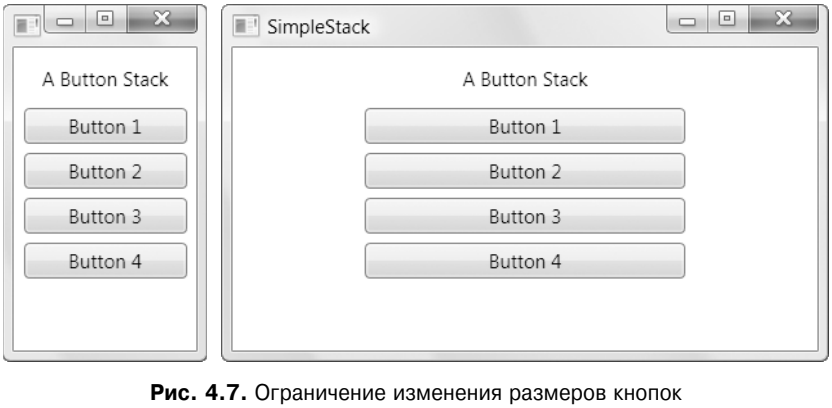

Рис. 4.7. Ограничение изменения размеров кнопок

Слева представлено окно в минимальном размере. Кнопки имеют размер по 100 единиц каждая, и окно не может быть сужено, чтобы сделать их меньше. Если вы попытаетесь сжать окно от этой точки, то правая часть каждой кнопки будет усечена. (Вы можете предотвратить такую возможность применением свойства `MinWidth` к самому окну, так что окно нельзя будет сузить меньше минимальной ширины.)

При увеличении окна кнопки также растут, пока не достигнут своего максимума в 200 единиц. Если после этого вы продолжите увеличивать окно, то с каждой стороны от кнопок будет добавлено дополнительное пространство (как показано на рисунке справа).

На заметку! В некоторых ситуациях вы можете использовать код, проверяющий, насколько велик элемент в окне. Свойства `Height` и `Width` не помогают, потому что указывают желаемые установки размера, которые могут не соответствовать действительному визуализируемому размеру. В идеальном сценарии вы позволите размерам ваших элементов вмещать их содержимое, и тогда свойства `Height` и `Width` вообще устанавливать не надо. Однако вы можете узнать действительный размер, используемый для визуализации элемента, прочитав свойства `ActualHeight` и `ActualWidth`. Но помните, что эти значения могут меняться при изменении размера окна или содержимого элементов.

Окна с автоматически устанавливаемыми размерами

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

- Во многих случаях вы хотите сделать окно меньше, чем диктует желаемый размер его дочерних элементов. Например, если ваше окно включает контейнер с прокручиваемым текстом, вы захотите ограничить размер этого контейнера, чтобы прокрутка была возможна. Вы не захотите показывать это окно нелепо большим, так чтобы не было потребности в прокрутке, чего требует контейнер. (Подробнее о прокрутке вы узнаете из главы 5.)
- Минимальный размер окна может быть удобен, но при этом не иметь наиболее привлекательных пропорций. Некоторые размеры окна просто лучше выглядят.
- Автоматическое изменение размеров окна не ограничено размером дисплея вашего монитора. Так что окно с автоматически установленным размером может оказаться слишком большим для просмотра.

Однако окна с автоматически устанавливаемым размером возможны, и они имеют смысл, когда вы конструируете простое окно с динамическим содержимым. Чтобы включить автоматическую установку размеров окна, удалите свойства `Height` и `Width` и установите `Window.SizeToContent` равным `WidthAndHeight`. Окно сделает себя достаточно большим, чтобы вместить его содержимое. Вы можете также позволить окну изменять свой размер только в одном измерении, используя значение `SizeToContent` для `Width` или `Height`.

WrapPanel и DockPanel

Очевидно, что только `StackPanel` не может помочь вам в создании реалистичного пользовательского интерфейса. Чтобы довершить картину, `StackPanel` должен работать с другими более развитыми контейнерами компоновки. Только так вы сможете создать полноценное окно.

Наиболее изощренный контейнер компоновки — это `Grid`, который мы рассмотрим далее в этой главе. Но сначала стоит взглянуть на `WrapPanel` и `DockPanel` — два самых

простых контейнера компоновки, предоставленных WPF. Они дополняют `StackPanel` другим поведением компоновки.

WrapPanel

`WrapPanel` располагает элементы управления в доступном пространстве — по одной строке или колонке за раз. По умолчанию `WrapPanel.Orientation` устанавливается в `Horizontal`; элементы управления располагаются слева направо, затем — в следующих строках. Однако вы можете использовать `Vertical` для размещения элементов в нескольких колонках.

Совет. Подобно `StackPanel`, панель `WrapPanel` действительно предназначена для управления мелкими деталями пользовательского интерфейса, а не компоновкой всего окна. Например, вы можете 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>
```

На рис. 4.8 показано, как переносятся кнопки, чтобы заполнить текущий размер `WrapPanel` (определяемый размером окна, содержащего его). 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`. В примере, представленном слева на рис. 4.8, все кнопки выстроены в одну строку, причем некоторые растянуты, а другие выровнены по этой строке. В примере справа несколько кнопок выталкиваются на вторую строку. Поскольку вторая строка не включает слишком высоких кнопок, высота строки равна минимальной высоте кнопок. В результате не имеет значения, какую установку `VerticalAlignment` используют кнопки в этой строке.

На заметку! `WrapPanel` — единственная панель, которая не может дублироваться хитрым использованием `Grid`.

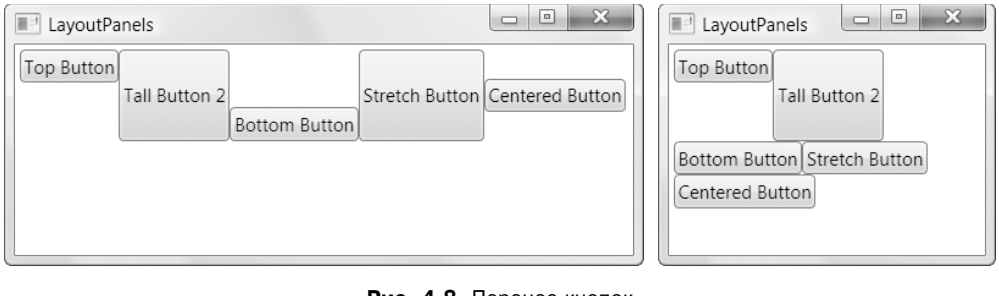

Рис. 4.8. Перенос кнопок