ГЛАВА 10

Ресурсы

Система ресурсов WPF — это просто способ поддержания вместе набора полезных объектов, таких как часто используемые кисти, стили или шаблоны, существенно упрощающий работу с ними.

Несмотря на возможность создания и манипулирования ресурсами в коде, обычно они определяются в XAML-разметке. Как только ресурс определен, его можно применять повсюду в остальной разметке окна (а в случае ресурса приложения — везде в остальной части приложения). Такой подход упрощает разметку, сокращает количество повторяющихся фрагментов кода и позволяет хранить детали, касающиеся пользовательского интерфейса (вроде цветовой схемы приложения), в центральном месте, чтобы в дальнейшем их было проще модифицировать. Объектные ресурсы также служат основой для многократного использования стилей WPF, как будет показано в следующей главе.

На заметку! Не путайте объектные ресурсы WPF с ресурсами сборки, которые рассматривались в главе 7. Ресурс сборки — это порция двоичных данных, которая встраивается в скомпилированную сборку. Ресурс сборки можно применять для обеспечения приложения необходимым изображением или звуковым файлом. Объектный ресурс, с другой стороны, представляет собой объект .NET, который нужно определить в одном месте, а использовать во множестве других.

Общие сведения о ресурсах

Инфраструктура WPF позволяет определять ресурсы в коде или в различных местах разметки (вместе с конкретными элементами управления, внутри отдельных окон или же во всем приложении).

Ресурсы обладают рядом важных преимуществ, которые перечислены ниже.

- *Эффективность.* Ресурсы позволяют определять объект один раз и затем использовать его в нескольких местах внутри разметки. Это упрощает код и делает его намного эффективнее.
- *Сопровождаемость.* Ресурсы позволяют переносить низкоуровневые детали форматирования (такие как размеры шрифтов) в центральное место, где их легко изменять. Это своего рода XAML-эквивалент создания констант в коде.
- *Адаптируемость.* После отделения определенной информации от остальной части приложения и ее помещения в раздел ресурсов появляется возможность динамической модификации этой информации. Например, может понадобиться изменять детали ресурсов на основе пользовательских предпочтений или текущего языка.

Коллекция ресурсов

Каждый элемент включает свойство Resources, в котором хранится словарная коллекция ресурсов (представляющая собой экземпляр класса ResourceDictionary). Эта коллекция ресурсов может хранить объект любого типа с индексацией по строке.

Хотя каждый элемент имеет свойство Resources (которое определено в классе FrameworkElement), чаще всего ресурсы определяются на уровне окна. Причина в том, что каждый элемент имеет доступ к ресурсам из собственной коллекции ресурсов, а также к ресурсам из коллекций ресурсов всех своих родительских элементов.

Например, рассмотрим окно с тремя кнопками, показанное на рис. 10.1. Две из трех кнопок используют одну и ту же кисть — кисть изображения, которая закрашивает их мозаичным узором с улыбающимися рожицами.

Рис. 10.1. Окно с повторно используемой кистью

В данном случае вполне очевидно, что верхняя и нижняя кнопки должны иметь одинаковый стиль. Однако не исключено, что позже может понадобиться изменить какие-то характеристики кисти изображения. По этой причине имеет смысл определить данную кисть в ресурсах окна и затем при необходимости просто повторно использовать ее.

Ниже показано, как определяется кисть:

```
<Window.Resources>
  <ImageBrush x:Key="TileBrush" TileMode="Tile"
   ViewportUnits="Absolute" Viewport="0 0 32 32"
   ImageSource="happyface.jpg" Opacity="0.3">
  </ImageBrush>
</Window.Resources>
```

Детали кисти изображения здесь не имеют особого значения (хотя в главе 12 эти вопросы рассматриваются более подробно). По-настоящему *важную* роль играет первый атрибут Key (предваренный префиксом пространства имен x, который обеспечивает размещение в пространстве имен XAML, а не WPF). В нем указано имя, по которому кисть будет индексироваться в коллекции Window.Resources. Можно использовать любое имя — главное, чтобы оно же было указано при извлечении ресурса.

На заметку! В разделе ресурсов можно создавать экземпляр любого класса .NET (в том числе собственных специальных классов), который является дружественным к XAML. Это означает, что он должен обладать несколькими базовыми характеристиками, такими как наличие открытого конструктора без аргументов и свойств, допускающих запись.

Чтобы использовать ресурс в XAML-разметке, необходим какой-то способ ссылки на него. Это достигается с использованием расширения разметки. В действительности доступны два расширения разметки: одно предназначено для динамических ресурсов, а второе — для статических ресурсов. Статические ресурсы устанавливаются один раз, при первом создании окна. Динамические ресурсы применяются повторно в случае изменения ресурса. (Различия между этими типами ресурсов более подробно рассматриваются далее в главе.) В этом примере кисть изображения никогда не изменяется, поэтому статический ресурс является вполне подходящим вариантом.

Ниже приведена разметка одной из кнопок, использующих данный ресурс:

```
<Button Background="{StaticResource TileBrush}"
 Margin="5" Padding="5" FontWeight="Bold" FontSize="14">
  A Tiled Button
</Button>
```

В этом случае ресурс извлекается и присваивается свойству `Button.Background`. То же самое можно проделать (со слегка большими накладными расходами) за счет применения динамического ресурса:

```
<Button Background="{DynamicResource TileBrush}"
```

Использовать для ресурса простой объект .NET действительно просто. Однако существует несколько нюансов, которые должны быть учтены. О них пойдет речь в следующих разделах.

Иерархия ресурсов

Каждый элемент имеет собственную коллекцию ресурсов, и WPF производит рекурсивный поиск необходимого ресурса в дереве элементов. Благодаря этому, в текущем примере кисть изображения можно было бы перенести из коллекции `Resources` окна в коллекцию `Resources` содержащего все три кнопки элемента `StackPanel`, не изменяя способ работы приложения. Кисть изображения также можно было бы переместить в коллекцию `Button.Resources`, но тогда потребовалось бы определять ее для каждой кнопки.

Существует еще одна особенность, которую следует учесть. Статический ресурс всегда должен быть определен в коде разметки *перед* ссылкой на него. Это означает, что хотя размещение раздела `Windows.Resources` после основного контента окна (панели `StackPanel`, содержащей все кнопки) вполне допустимо, такое изменение приведет к нарушению работоспособности текущего примера. Встретив статическую ссылку на неизвестный ресурс, анализатор XAML сгенерирует исключение. (Эту проблему можно обойти за счет использования динамического ресурса, но веские причины для внесения дополнительных накладных расходов отсутствуют.)

В результате, если необходимо поместить ресурс в элемент кнопки, код разметки должен быть немного перестроен, чтобы ресурс определялся до установки фона. Ниже показан один из возможных способов:

```
<Button Margin="5" Padding="5" FontWeight="Bold" FontSize="14">
  <Button.Resources>
    <ImageBrush x:Key="TileBrush" TileMode="Tile"
     ViewportUnits="Absolute" Viewport="0 0 10 10"
     ImageSource="happyface.jpg" Opacity="0.3"></ImageBrush>
  </Button.Resources>
  <Button.Background>
    <StaticResource ResourceKey="TileBrush"/>
  </Button.Background>
  <Button.Content>Another Tiled Button</Button.Content>
</Button>
```

Синтаксис для расширения разметки статического ресурса в этом примере выглядит немного по-другому, поскольку устанавливается во вложенном элементе (а не в атрибуте). Ключ ресурса, указывающий на правильный ресурс, задается с применением свойства ResourseKey.

Интересно отметить, что имена ресурсов можно использовать повторно, главное — не применять одно и то же имя более одного раза в рамках одной и той же коллекции. Это означает, что окно можно было бы создать и с помощью показанного ниже кода, где кисть изображения определяется в двух местах:

```
<Window x:Class="Resources.TwoResources"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Resources" Height="300" Width="300" >

  <Window.Resources>
    <ImageBrush x:Key="TileBrush" TileMode="Tile"
            ViewportUnits="Absolute" Viewport="0 0 32 32"
            ImageSource="happyface.jpg" Opacity="0.3"></ImageBrush>
  </Window.Resources>

  <StackPanel Margin="5">
    <Button Background="{StaticResource TileBrush}" Padding="5"
     FontWeight="Bold" FontSize="14" Margin="5" >A Tiled Button</Button>

    <Button Padding="5" Margin="5"
     FontWeight="Bold" FontSize="14">A Normal Button</Button>
      <Button Background="{DynamicResource TileBrush}" Padding="5" Margin="5"
       FontWeight="Bold" FontSize="14">
      <Button.Resources>
        <ImageBrush x:Key="TileBrush" TileMode="Tile"
         ViewportUnits="Absolute" Viewport="0 0 32 32"
         ImageSource="sadface.jpg" Opacity="0.3"></ImageBrush>
      </Button.Resources>
      <Button.Content>Another Tiled Button</Button.Content>
    </Button>

  </StackPanel>
</Window>
```

В данном случае кнопка использует ресурс, найденный первым. Поскольку поиск начинается с собственной коллекции Resources, вторая кнопка использует изображение sadface.jpg, в то время как первая кнопка извлекает кисть из содержащего окна и работает с изображением happyface.jpg.

Статические и динамические ресурсы

Из-за того, что в предыдущем примере использовался статический ресурс (в роли которого выступала кисть изображения), могло сложиться впечатление, что статический ресурс невосприимчив ни к каким изменениям. Однако это не так.

Например, предположим, что на каком-то этапе после применения ресурса и отображения окна выполняется следующий код:

```
ImageBrush brush = (ImageBrush)this.Resources["TileBrush"];
brush.Viewport = new Rect(0, 0, 5, 5);
```

Этот код извлекает кисть из коллекции Window.Resources и затем определенным образом модифицирует ее. (Формально код изменяет размер каждого фрагмента кисти, сжимая улыбающуюся рожицу и теснее упаковывая узор.) Может показаться, что никакой реакции в пользовательском интерфейсе после выполнения этого кода быть не должно — в конце концов, это же статический ресурс. Тем не менее, изменение распро-

страняется на две кнопки. Фактически кнопки обновляются и получают новое значение для свойства Viewport независимо от того, используют они кисть через статический или через динамический ресурс.

Причина в том, что класс Brush является производным от класса Freezable. Класс Freezable обладает средствами для отслеживания базовых изменений (и может быть “заморожен” в доступном только для чтения состоянии, если он не нуждается в изменении). Это означает, что при каждом изменении кисти в WPF все использующие эту кисть элементы управления обновляются автоматически. Не имеет значения, получают они эту кисть через ресурс или нет.

На этом этапе, скорее всего, возникает вопрос: чем же тогда отличаются статические и динамические ресурсы? Отличие заключается в том, что статический ресурс извлекает объект из коллекции ресурсов только один раз. В зависимости от типа объекта (и способа его использования) любые вносимые в этот объект изменения могут быть отмечены сразу же. Однако динамический ресурс ищет объект в коллекции ресурсов каждый раз, когда в нем возникает необходимость. Это означает, что под тем же самым ключом может размещаться совершенно новый объект, и динамический ресурс будет подхватывать это изменение.

Чтобы проиллюстрировать это отличие на примере, рассмотрим следующий код, который заменяет текущую кисть изображения совершенно новой кистью с однотонной заливкой голубого цвета:

```
this.Resources["TileBrush"] = new SolidColorBrush(Colors.LightBlue);
```

Динамический ресурс подхватит это изменение, а статический не будет иметь ни малейшего понятия о том, что его кисть была заменена в коллекции Resources какой-то другой. Он продолжит пользоваться исходной кистью ImageBrush.

На рис. 10.2 показан этот пример в окне, включающем динамический ресурс (верхняя кнопка) и статический ресурс (нижняя кнопка).

Обычно накладные расходы, связанные с использованием динамического ресурса, не особо приемлемы, и приложение будет прекрасно работать со статическим ресурсом. Одним заметным исключением является ситуация, когда создаются ресурсы, зависящие от настроек Windows (например, системных цветов). В этой ситуации должны применяться динамические ресурсы, чтобы приложение могло реагировать на любое изменение в текущей цветовой схеме. (В случае использования статических ресурсов приложение будет работать со старой цветовой схемой до тех пор, пока пользователь его не перезапустит.) Более подробно об этом речь пойдет далее в главе.

Рис. 10.2. Динамический и статический ресурсы

Динамические свойства рекомендуется использовать только при соблюдении перечисленных ниже условий:

- ресурс имеет свойства, которые зависят от настроек системы (таких как текущие цвета или шрифты Windows);
- планируется заменять объекты ресурсов программным образом (например, для реализации средства динамических обложек, как будет показано в главе 17).

Однако не стоит чрезмерно увлекаться динамическими ресурсами. Главная проблема в том, что изменение ресурса не обязательно приводит к обновлению пользовательского интерфейса. (В приведенном примере с кистью обновление происходит благодаря способу, которым конструируются объекты кисти — в частности потому, что они имеют встроенную поддержку уведомлений.) Если требуется динамическое обновление (другими словами, возможность изменения контента и автоматической самонастройки элементов управления), обычно гораздо удобнее применять привязку данных.

На заметку! В редких случаях динамические ресурсы также используются для ускорения первоначальной загрузки окна. Это объясняется тем, что статические ресурсы всегда загружаются при создании окна, в то время как динамические ресурсы загружаются при их первом использовании. Тем не менее, улучшения становятся заметными только в случае, если ресурс является исключительно большим и сложным (при этом синтаксический анализ его разметки занимает ощутимое время).

Неразделяемые ресурсы

Обычно когда ресурс используется во множестве мест, применяется один и тот же экземпляр объекта. Такое поведение — называемое *разделением* — как правило, оказывается тем, что нужно. Тем не менее, допустимо также указывать анализатору на необходимость создания отдельного экземпляра объекта при каждом его использовании.

Разделение отключается с помощью атрибута `Shared`:

```
<ImageBrush x:Key="TileBrush" x:Shared="False" ...></ImageBrush>
```

Для применения неразделяемых ресурсов существует несколько веских причин. Рассматривать вариант неразделяемых ресурсов стоит в ситуации, если позже планируется модифицировать экземпляры ресурса по отдельности. Например, можно было бы создать окно с несколькими кнопками, использующими одну и ту же кисть, но отключить разделение, чтобы иметь возможность изменять кисть каждой из кнопок индивидуально. Такой подход не особо распространен, поскольку неэффективен. В этом случае лучше позволить всем кнопкам использовать одну и ту же кисть изначально, а затем создавать и применять новые объекты кисти по мере необходимости. Тогда иметь дело с накладными расходами, связанными с дополнительными объектами кисти, придется только там, где это действительно необходимо.

Еще одна причина применения неразделяемых ресурсов — необходимость повторного использования объекта таким способом, который в противном случае невозможен. Например, это позволяет определить элемент (вроде `Image` или `Button`) как ресурс и затем отображать его в нескольких местах внутри окна.

Опять-таки, обычно это не самый лучший подход. Например, для повторного использования элемента `Image` целесообразнее сохранить соответствующий фрагмент информации (такой как объект `BitmapImage`, идентифицирующий источник изображения) и затем разделять его между несколькими элементами `Image`. Если просто нужно стандартизировать элементы управления так, чтобы они совместно использовали одни и те же свойства, гораздо лучше применять стили, которые подробно рассматривают-

ся в следующей главе. Стили предоставляют возможность создавать идентичные или почти идентичные копии любого элемента, а также переопределять значения свойств, когда они не подходят, и присоединять конкретные обработчики событий, чего нельзя делать в случае простого клонирования элемента, используя неразделяемый ресурс.

Доступ к ресурсам в коде

Обычно ресурсы определяются и используются в разметке. Однако при необходимости можно работать с коллекцией ресурсов и в коде.

Как уже было показано, элементы можно извлекать из коллекции ресурсов по имени. Однако при таком подходе должна использоваться коллекция ресурсов правильного элемента. Это ограничение не распространяется на разметку. Элемент управления, такой как кнопка, может извлекать ресурс, не зная точно, где тот определен. Когда свойству `Background` кнопки присваивается кисть, инфраструктура WPF проверяет коллекцию ресурсов кнопки на предмет наличия ресурса по имени `TileBrush`, затем коллекцию ресурсов содержащего ее элемента `StackPanel`, после чего коллекцию ресурсов содержащего окна. (В действительности, как будет показано в следующем разделе, этот процесс продолжается поиском в ресурсах приложения и системы.)

Искать ресурс подобным образом можно с помощью метода `FrameworkElement.FindResource()`. Ниже приведен пример, в котором при срабатывании события `Click` производится поиск ресурса кнопки (или одного из ее контейнеров более высокого уровня):

```
private void cmdChange_Click(object sender, RoutedEventArgs e)
{
    Button cmd = (Button)sender;
    ImageBrush brush = (ImageBrush)sender.FindResource("TileBrush");
    ...
}
```

Взамен `FindResource()` можно также использовать метод `TryFindResource()`, который вместо генерации исключения, если ресурс не найден, возвращает ссылку `null`.

Кстати, ресурсы можно также добавлять программно. Для этого необходимо выбрать элемент для размещения ресурса и вызвать метод `Add()` коллекции ресурсов. Однако гораздо чаще ресурсы определяются в разметке.

Ресурсы приложения

Элемент `Window` не является последним местом поиска ресурса. Если указан ресурс, который не удается найти ни в элементе управления, ни в одном из его контейнеров (вплоть до окна или страницы, содержащей этот элемент), WPF продолжает проверку в наборе ресурсов, которые были определены для приложения. В Visual Studio таковыми являются ресурсы, которые были определены в разметке внутри файла `App.xaml`:

```
<Application x:Class="Resources.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Menu.xaml"
    >
    <Application.Resources>
      <ImageBrush x:Key="TileBrush" TileMode="Tile"
        ViewportUnits="Absolute" Viewport="0 0 32 32"
        ImageSource="happyface.jpg" Opacity="0.3">
      </ImageBrush>
    </Application.Resources>
</Application>
```

Несложно догадаться, что ресурсы приложения предоставляют прекрасную возможность для многократного использования объекта во всем приложении. В приведенном примере неплохим вариантом будет применение кисти изображения в более чем одном окне.

На заметку! Прежде чем создавать ресурс приложения, оцените компромисс между сложностью и возможностью многократного использования. Добавление ресурса приложения предоставляет большую степень повторного использования, но увеличивает сложность, поскольку не позволяет сразу же понять, в каких окнах применяется данный ресурс. (Концептуально это напоминает программу C++ старого стиля, с чрезмерно большим количеством глобальных переменных.) Ресурсы приложения рекомендуется создавать, если объект многократно используется повсеместно в приложении (например, во множестве окон). Если же он используется только в двух или трех местах, тогда лучше определять ресурс в каждом окне.

Оказывается, ресурсы приложения *все еще* не являются последним местом при выполнении элементом поиска ресурса. Если обнаружить нужный ресурс в ресурсах приложения не удается, поиск продолжается в ресурсах системы.

Ресурсы системы

Как упоминалось ранее, динамические ресурсы в первую очередь предназначены для того, чтобы помочь приложению реагировать на изменения в системных настройках. Однако при этом сразу возникает вопрос: как извлечь настройки системы и работать с ними в коде? Секрет кроется в наборе из трех классов `SystemColors`, `SystemFonts` и `SystemParameters`, которые расположены в пространстве имен `System.Windows`. Класс `SystemColors` предоставляет доступ к настройкам цвета. Класс `SystemFonts` обеспечивает доступ к настройкам шрифтов. Класс `SystemParameters` охватывает огромный список настроек, которые описывают стандартный размер различных экранных элементов, параметры клавиатуры и мыши, размер экрана, а также то, включены ли разнообразные графические эффекты (такие как горячее отслеживание, отбрасывание теней и отображение контента окна при перетаскивании).

На заметку! Классы `SystemColors` и `SystemFonts` доступны в двух версиях, которые размещены в пространствах имен `System.Windows` и `System.Drawing`. Версии из `System.Windows` являются частью WPF. Они используют правильные типы данных и поддерживают систему ресурсов. Версии из `System.Drawing` относятся к инфраструктуре Windows Forms. В приложениях WPF они бесполезны.

Классы `SystemColors`, `SystemFonts` и `SystemParameters` открывают все свои детали через статические свойства. Например, свойство `SystemColors.WindowTextColor` предоставляет структуру `Color`, которую можно использовать любым желаемым образом. Ниже приведен пример ее применения для создания кисти и закрашивания переднего плана элемента:

```
label.Foreground = new SolidBrush(SystemColors.WindowTextColor);
```

Однако эффективнее воспользоваться просто готовым свойством кисти:

```
label.Foreground = SystemColors.WindowTextBrush;
```

Доступ к статическим свойствам в WPF можно получать с помощью статического расширения разметки. Например, вот как установить основной фон того же самого элемента `label` в XAML-разметке:

```
<Label Foreground="{x:Static SystemColors.WindowTextBrush}">
  Ordinary text
</Label>
```

В этом примере не используется ресурс. Кроме того, с ним связан один небольшой недостаток: при выполнении синтаксического анализа окна и создании метки кисть создается на основании текущего “снимка” цвета текста в окне. В случае изменения цветов Windows во время работы приложения (после отображения окна, содержащего метку) метка не будет обновлена. Приложения, которые ведут себя подобным образом, считаются несколько грубыми.

Решить эту проблему указанием в свойстве `Foreground` непосредственно объекта кисти невозможно. Вместо этого `Foreground` необходимо установить в объект `DynamicResource`, служащий оболочкой для данного системного ресурса. К счастью, все классы `System`*`Xxx`* предоставляют дополняющий набор свойств, которые возвращают объекты `ResourceKey` — ссылки, позволяющие извлекать ресурс из коллекции ресурсов системы. Эти свойства имеют те же имена, что и обычные свойства, напрямую возвращающие объект, со словом `Key` в конце. Например, ключом ресурса для `SystemColors.WindowTextBrush` будет `SystemColors.WindowTextBrushKey`.

На заметку! Ключи ресурсов представляют собой не простые имена, а ссылки, которые сообщают WPF о том, где следует искать конкретный ресурс. Класс `ResourceKey` является непрозрачным, т.е. он не показывает низкоуровневые детали о том, как идентифицируются ресурсы системы. Однако переживать о возможных конфликтах между собственными ресурсами и ресурсами системы не нужно, поскольку они находятся в отдельных сборках и трактуются по-разному.

Ниже показано, как использовать ресурс из классов `System`*`Xxx`*:

```
<Label Foreground="{DynamicResource {x:Static SystemColors.WindowTextBrushKey}}">
  Ordinary text
</Label>
```

Этот код разметки немного сложнее приведенного в предыдущем примере. Он начинается с определения динамического ресурса. Тем не менее, динамический ресурс не извлекается из коллекции ресурсов в приложении. Вместо этого применяется ключ, определенный свойством `SystemColors.WindowTextBrushKey`. Поскольку свойство является статическим, также должно использоваться статическое расширение разметки, чтобы анализатор смог понять намерения.

После внесения такого изменения метка будет без проблем обновляться в случае изменения настроек системы.

Словари ресурсов

Для разделения ресурсов между множеством проектов можно создать *словарь ресурсов*. Словарь ресурсов — это просто XAML-документ, который всего лишь хранит необходимые ресурсы.

Создание словаря ресурсов

Ниже приведен пример словаря ресурсов с одним ресурсом внутри:

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

  <ImageBrush x:Key="TileBrush" TileMode="Tile"
   ViewportUnits="Absolute" Viewport="0 0 32 32"
   ImageSource="happyface.jpg" Opacity="0.3">
  </ImageBrush>
</ResourceDictionary>
```

При добавлении словаря ресурсов в приложение удостоверьтесь, что свойство Build Action (Действие построения) установлено в Page (страница), как принято для любого другого XAML-файла. Это обеспечит компиляцию словаря ресурсов в формат BAML для достижения более высокой производительности. Тем не менее, вполне допустимо установить свойство Build Action словаря ресурсов в Resource (ресурс); в этом случае ресурс будет встраиваться в сборку, но не компилироваться. В результате синтаксический анализ во время выполнения будет происходить несколько медленнее.

Использование словаря ресурсов

Чтобы использовать словарь ресурсов, где-то в приложении его необходимо объединить с коллекцией ресурсов. Это можно было бы сделать в конкретном окне, однако чаще объединение осуществляется на уровне коллекции ресурсов приложения, как показано ниже:

```
<Application x:Class="Resources.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Menu.xaml" >
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="AppBrushes.xaml"/>
        <ResourceDictionary Source="WizardBrushes.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

В приведенной разметке объект `ResourceDictionary` создается явно. Коллекция ресурсов всегда представляет собой объект `ResourceDictionary`, но в данном случае эта деталь должна быть указана явно, чтобы иметь возможность также устанавливать свойство `ResourceDictionary.MergedDictionaries`. Если не предпринять этот шаг, значением свойства `MergedDictionaries` будет `null`.

Коллекция `MergedDictionaries` содержит объекты `ResourceDictionary`, которые будут использоваться для пополнения коллекции ресурсов. В рассматриваемом случае объектов `ResourceDictionary` два: первый определен в словаре ресурсов `AppBrushes.xaml`, а второй — в `WizardBrushes.xaml`.

Чтобы добавить собственные ресурсы *и* объединить их со словарями ресурсов, необходимо просто поместить их до или после раздела `MergedProperties`:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="AppBrushes.xaml"/>
      <ResourceDictionary Source="WizardBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
    <ImageBrush x:Key="GraphicalBrush1" ...="" ></ImageBrush>
    <ImageBrush x:Key="GraphicalBrush2" ...="" ></ImageBrush>
  </ResourceDictionary>
</Application.Resources>
```

На заметку! Как упоминалось ранее, вполне разумно хранить ресурсы с одинаковыми именами в разных, но перекрывающихся коллекциях ресурсов. Однако не допускается объединять словари ресурсов, которые содержат ресурсы с одинаковыми именами. При обнаружении дубликата во время компиляции приложения генерируется исключение `XamlParseException`.

Одна из причин использования словарей ресурсов — определение одной или нескольких многократно используемых “обложек” приложения, которые можно применять к элементам управления (об этом речь пойдет в главе 17). Еще одна причина связана с необходимостью хранения контента, который должен быть локализован (такого как строки сообщений об ошибках).

Разделение ресурсов между сборками

Если необходимо использовать словарь ресурсов во множестве приложений, можно копировать и распространять содержащий его XAML-файл. Это самый простой подход, но он не поддерживает управление версиями. Более структурированный прием предусматривает компиляцию словаря ресурсов в отдельную сборку типа библиотеки классов и распространение его в виде такого компонента.

При разделении скомпилированной сборки с одним или несколькими словарями ресурсов возникает еще одна трудность, а именно — нужен какой-то способ для извлечения необходимого ресурса и его использования в приложении. Здесь возможны два варианта. Самым простым решением является применение кода, который создает соответствующий объект ResourceDictionary. Например, если словарь ресурсов находится в сборке типа библиотеки классов по имени ReusableDictionary.xaml, для его создания вручную можно воспользоваться следующим кодом:

```
ResourceDictionary resourceDictionary = new ResourceDictionary();
resourceDictionary.Source = new Uri(
  "ResourceLibrary;component/ReusableDictionary.xaml", UriKind.Relative);
```

В этом фрагменте кода применяется синтаксис URI типа “pack”, который рассматривался ранее. Код конструирует относительный URI, указывающий на скомпилированный XAML-ресурс по имени ReusableDictionary.xaml, который расположен в другой сборке. После создания объекта ResourceDictionary необходимый ресурс можно извлекать из коллекции вручную:

```
cmd.Background = (Brush)resourceDictionary["TileBrush"];
```

Однако назначать ресурсы вручную не понадобится. После загрузки нового словаря ресурсов любые имеющиеся в окне ссылки DynamicResource будут автоматически оцениваться заново. Пример применения этого приема будет приведен в главе 17 во время построения средства динамического снабжения обложками.

Для тех, кто не желает писать никакого кода, доступен еще один вариант. Можно использовать специально предназначенное для этой цели расширение разметки ComponentResourceKey. Оно применяется для создания имени ключа ресурса. Данное расширение указывает WPF, что ресурс планируется разделять между сборками.

На заметку! Вплоть до этого момента демонстрировались только ресурсы, у которых для имен ключей использовались строки (наподобие “TileBrush”). Строки являются самым типичным способом именования ресурсов. Тем не менее, WPF обладает интеллектуальной функцией расширяемости ресурсов, которая активизируется автоматически, когда в качестве имен ключей применяются некоторые типы, не являющиеся строками. Например, в следующей главе будет показано, как использовать объект Type для имени ключа стиля. Это укажет WPF автоматически применять данный стиль к элементам соответствующего типа. Аналогично экземпляр ComponentResourceKey можно использовать в качестве имени ключа для любого ресурса, который должен разделяться между сборками.

Прежде чем двигаться дальше, необходимо позаботиться о назначении словарю ресурсов корректного имени. Для этого словарь ресурсов должен находиться в файле по имени generic.xaml, а этот файл — в подпапке Themes приложения. Ресурсы в файлах

generic.xaml воспринимаются как часть стандартной темы и потому всегда делаются доступными. Этот прием будет встречаться еще не раз, в частности, во время построения специальных элементов управления в главе 18.

На рис. 10.3 показано, как выглядит надлежащая организация файлов. Верхний проект, именуемый ResourceLibrary, включает файл generic.xaml в корректной папке. Нижний проект, имеющий имя Resources, включает ссылку на проект ResourceLibrary и потому может пользоваться содержащимися в нем ресурсами.

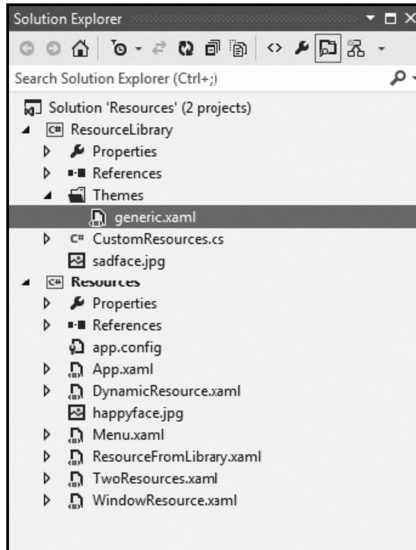

Рис. 10.3. Разделение ресурсов с помощью библиотеки классов

Совет. Чтобы организовать наилучшим образом большое количество ресурсов, можно создавать отдельные словари ресурсов так, как было показано ранее. Однако эти словари должны быть обязательно включены в файл generic.xaml, чтобы к ним можно было получать доступ.

Следующий шаг заключается в создании имени ключа для разделяемого ресурса, который хранится в сборке ResourceLibrary. В случае использования ComponentResourceKey необходимо предоставить два фрагмента информации — ссылку на соответствующий класс в сборке библиотеки классов и описательный идентификатор ресурса. Ссылка на класс — это часть той “магии”, которая позволяет WPF разделять ресурс с другими сборками. При использовании этого ресурса сборки будут предоставлять ту же самую ссылку на класс и тот же самый идентификатор ресурса.

То, как класс выглядит на самом деле, роли не играет, и он не обязан содержать код. Сборка, в которой определен данный тип — это та же самая сборка, где ComponentResourceKey будет искать ресурс. В примере, показанном на рис. 10.3, используется класс CustomResources, который не содержит кода:

```
public class CustomResources
{}
```

Теперь можно создать имя ключа с использованием этого класса и идентификатора ресурса:

```
x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:CustomResources},
  ResourceId=SadTileBrush}"
```

Ниже приведена полная разметка для файла generic.xaml, которая включает единственный ресурс ImageBrush, использующий другое графическое изображение:

```
<ResourceDictionary
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:local="clr-namespace:ResourceLibrary">

  <ImageBrush
   x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:CustomResources},
ResourceId=SadTileBrush}"
   TileMode="Tile" ViewportUnits="Absolute" Viewport="0 0 32 32"
   ImageSource="ResourceLibrary;component/sadface.jpg" Opacity="0.3">
  </ImageBrush>
</ResourceDictionary>
```

Внимательные читатели заметят в этом примере одну неожиданную деталь — свойство ImageSource больше не устанавливается с использованием имени изображения (sadface.jpg). Взамен применяется более сложный относительный URI, который четко указывает, что изображение является частью компонента ResourceLibrary. Это обязательный шаг, потому что данный ресурс будет использоваться в контексте другого приложения. Если указать просто имя изображения, то приложение будет искать изображение только в собственных ресурсах. Именно по этой причине необходим относительный URI, который указывает компонент, где хранится изображение.

Теперь, когда словарь ресурсов создан, его можно использовать в другом приложении. Для начала нужно позаботиться об определении префикса для сборки библиотеки классов, как показано ниже:

```
<Window x:Class="Resources.ResourceFromLibrary"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:res="clr-namespace:ResourceLibrary;assembly=ResourceLibrary"
 ... >
```

Затем можно применять объект DynamicResource, содержащий ComponentResourceKey. (В этом есть смысл, поскольку ComponentResourceKey *является* именем ресурса.) Используемый потребителем ComponentResourceKey ничем не отличается от ComponentResourceKey из библиотеки классов. Вы предоставляете ссылку на тот же самый класс и тот же идентификатор ресурса. Единственное отличие заключается в том, что здесь нельзя применять тот же самый префикс пространства имен XML. В приведенном примере вместо префикса local используется res, чтобы подчеркнуть тот факт, что класс CustomResources определен в другой сборке:

```
<Button Background="{DynamicResource {ComponentResourceKey
 TypeInTargetAssembly={x:Type res:CustomResources}, ResourceId=SadTileBrush}}"
 Padding="5" Margin="5" FontWeight="Bold" FontSize="14">
  A Resource From ResourceLibrary
</Button>
```

На заметку! При использовании ComponentResourceKey должен применяться динамический, а не статический ресурс.

На этом пример завершен. Тем не менее, имеется еще один дополнительный шаг, который можно предпринять, чтобы упростить доступ к ресурсу. Можно определить статическое свойство, возвращающее корректный ключ ComponentResourceKey, который необходимо использовать. Обычно это свойство определяется в классе внутри компонента:

```
public class CustomResources
{
    public static ComponentResourceKey SadTileBrushKey
    {
        get
        {
            return new ComponentResourceKey(
              typeof(CustomResources), "SadTileBrush");
        }
    }
}
```

Теперь можно воспользоваться расширением разметки `Static`, чтобы получить доступ к этому свойству и применить ресурс, не указывая длинное значение `ComponentResourceKey` в разметке:

```
<Button
  Background="{DynamicResource {x:Static res:CustomResources.SadTileBrushKey}}"
  Padding="5" Margin="5" FontWeight="Bold" FontSize="14">
   A Resource From ResourceLibrary
</Button>
```

Этот удобный сокращенный подход представляет собой, по сути, тот же прием, который применялся в описанных ранее классах `SystemXxx`. Например, в результате извлечения `SystemColors.WindowTextBrushKey` получается корректный объект ключа ресурса. Единственное отличие состоит в том, что он является экземпляром закрытого класса `SystemResourceKey`, а не `ComponentResourceKey`. Оба эти класса являются производными от одного и того же предка — абстрактного класса по имени `ResourceKey`.

Резюме

В этой главе было показано, что система ресурсов WPF позволяет многократно использовать одни и те же объекты в различных частях приложения. Вы узнали, как объявлять ресурсы в коде и разметке, как применять ресурсы системы и как разделять ресурсы между множеством приложений с помощью сборок, представляющих собой библиотеки классов.

На этом рассмотрение ресурсов не завершается. Объектные ресурсы очень часто используются для хранения *стилей* — коллекций настроек для свойств, которые могут применяться к множеству элементов. Об определении стилей, сохранении их в виде ресурсов и повторном их использовании речь пойдет в следующей главе.