В. И. ЛОСКУТОВ, И. Л. КОРОБОВА

РАЗРАБОТКА ИНФОРМАЦИОННЫХ СИСТЕМ ДЛЯ WINDOWS STORE

Тамбов
• Издательство ФГБОУ ВПО «ТГТУ» •
2014

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Тамбовский государственный технический университет»

В. И. ЛОСКУТОВ, И. Л. КОРОБОВА

РАЗРАБОТКА ИНФОРМАЦИОННЫХ СИСТЕМ ДЛЯ WINDOWS STORE

Утверждено Учёным советом университета
в качестве учебного пособия для бакалавров направлений
подготовки 230100 «Информатика и вычислительная техника»,
080700 «Бизнес-информатика», 230400 «Информационные системы и
технологии» очной формы обучения

Тамбов
Издательство ФГБОУ ВПО «ТГТУ»
2014

УДК 004.4′236(078.8)
ББК 3973.04я73
Л79

Р е ц е н з е н т ы :

Кандидат технических наук, доцент
кафедры «Компьютерное и математическое моделирование»
ФГБОУ ВПО «ТГУ им. Г. Р. Державина»
В. П. Дудаков

Кандидат технических наук, профессор
кафедры «Информационные системы и защита информации»
ФГБОУ ВПО «ТГТУ»
Ю. Ф. Мартемьянов

Л79 **Лоскутов, В. И.**
Разработка информационных систем для Windows Store : учебное пособие для бакалавров направлений подготовки 230100, 080700, 230400 / В. И. Лоскутов, И. Л. Коробова. – Тамбов : Изд-во ФГБОУ ВПО «ТГТУ», 2014. – 80 с. – 100 экз. – ISBN 978-5-8265-1285-2.

Рассматриваются основные вопросы разработки информационных систем. Даётся описание процесса проектирования приложений, описываются процессы жизненного цикла. Приводятся требования к интерфейсу, содержанию. Описываются процессы реализации программной системы, в том числе гибридной (с использованием различных языков программирования).

Предназначено для бакалавров направлений подготовки 230100 «Информатика и вычислительная техника», 080700 «Бизнес-информатика», 230400 «Информационные системы и технологии» очной формы обучения

УДК 004.4′236(078.8)
ББК 3973.04я73

ISBN 978-5-8265-1285-2

ПРЕДИСЛОВИЕ

На любом этапе развития вычислительной техники мощные (на момент создания) вычислительные комплексы были недоступны простому пользователю. Ещё в 70-х годах прошлого века появились первые терминальные ЭВМ, которые использовали вычислительные ресурсы центральной вычислительной машины. Позже появилось понятие сервера – центрального узла вычислительной сети, который выполнял определённый набор функций и фактически управлял клиентскими машинами. Со временем производительность клиентских машин приблизилась к производительности серверов, за исключением СуперЭВМ, менфреймов и т.п. Тем не менее при наличии достаточно производительного клиента часть вычислительных задач перешла с сервера на клиентскую ЭВМ.

Сегодняшнему уровню развития вычислительной техники присущи три особенности:

1) весьма значительные объёмы обрабатываемой информации как у корпоративных пользователей, так и у частных;

2) повсеместное наличие высокоскоростных каналов связи и в первую очередь доступа в Internet;

3) появление новых классов устройств, различных по своим функциям и параметрам, в широком доступе, таких как смартфоны, планшеты, ноутбуки.

Эти особенности определили повторное бурное развитие уже известной по прошлому веку технологии: центральная ЭВМ – терминал. Однако реализация происходит на совершенно ином уровне. Центральная ЭВМ на сегодняшний день – это удалённый вычислительный центр, который доступен пользователю по каналам Internet. Причём количество пользователей, одновременно обслуживаемых, – это десятки (а в некоторых случаях сотни) тысяч. Предоставляются различные сервисы: от хранения телефонных контактов до сложных вычислительных задач. Такие технологии получили название «облако». Клиентами являются различные устройства пользователей как корпоративных, так и частных.

ВВЕДЕНИЕ

В широком смысле информационная система (ИС) есть совокупность технического, программного и организационного обеспечения, а также персонала, предназначенная для того, чтобы своевременно обеспечивать надлежащих людей надлежащей информацией. Основной задачей ИС является удовлетворение конкретных информационных потребностей в рамках конкретной предметной области.

В рамках данного курса будут рассматриваться технологии программного обеспечения Microsoft для архитектуры Windows RT.

В новой операционной системе Windows 8 представлен новый тип программы – приложение для Windows Store – магазина Windows. Приложения магазина Windows отличаются совершенно новым внешним видом и удобством использования. Их можно запускать на различных устройствах и продавать (распространять) в магазине Windows.

Все нововведения в Windows 8 могут быть описаны двумя фразами:

1) архитектура WinRT;
2) интерфейс Metro.

Несмотря на все различия, новая система использует значительное количество уже известных технологий. Основными являются технологии, основанные на платформе .NET, такие как Windows Presentation Foundation, ASP.NET, Silverlight и др.

Windows 8 является логическим продолжением Windows 7, поддерживая все возможности последней и дополняя их функционалом, ориентированным на новые требования рынка.

Новый интерфейс операционной системы (ОС), который и получил название Metro, позволяет работать как с помощью стандартных устройств (клавиатуры и мыши), так и с помощью прикосновений и жестов (см. рисунок).

Если запустить Windows 8, то на основном экране пользователь может увидеть набор анимированных плиток (tiles) вместо статического рабочего стола. Плитки располагаются в несколько рядов вдоль всего экрана, поддерживающего горизонтальный скроллинг.

Данные плитки легко перемещать в любое место экрана, просматривать на них динамическую информацию, а также менять их размер (поддерживаются два режима отображения – стандартный и расширенный, переключиться между которыми можно, щёлкнув плитку правой кнопкой мыши).

Стартовый экран Windows

Подобный интерфейс похож на основной экран в Windows Phone, предполагающий активную работу с помощью жестов. Интейфейс Metro был анонсирован в Windows Phone.

Естественно, задача разработчика состоит в том, чтобы не просто создать, но и, добавив анимацию к плитке своего приложения, придать ему некоторую уникальность.

После запуска одного из приложений пользователя ждёт сюрприз – полное отсутствие выделенного окна приложения. Это связано с тем, что в Windows 8 основная ставка сделана на представление контента и возможность пользователя работать с этим контентом, не отвлекаясь на другие вещи. Поэтому все приложения для Windows Store запускаются в полноэкранном режиме. И вот тут сюрприз, состоящий в разработке универсальных интерфейсов, подстерегает уже разработчика. Так, чтобы поддерживать работу приложения в полноэкранном режиме на всех типах устройств (с учётом разных разрешений, книжной или альбомной ориентации, работы с помощью жестов или клавиатуры), необходимо обладать большим запасом знаний и реализовывать дополнительный код.

Тут могут помочь знания XAML, полученные при создании приложений WPF или Silverlight. Правда, в последних разработчики часто устанавливали фиксированный размер окна.

Естественно, приложение может иметь меню и механизмы для установки настроек. Именно поэтому в Windows 8 появляются панель приложения (Application Bar) и системное меню. Если Вы работаете с клавиатурой, то панель приложения можно вызвать нажатием правой кнопки мыши (или жестом), а системное меню – перемещением курсора мыши в верхний правый угол экрана либо с помощью комбинации клавиш Win+C (или жеста).

1. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ОСНОВНЫЕ ЭТАПЫ СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С АРХИТЕКТУРОЙ WINRT

Новая операционная система использует известные технологии, в основе которых лежит платформа .NET: Windows Presentation Foundation, ASP.NET, Silverlight и др. В основе перечисленных технологий, так же как и многих других современных технологий разработки программного обеспечения (ПО) лежит подход, который основан на систематическом использовании моделей для языково-независимой разработки программной системы, на основе её прагматики.

Модель содержит не все признаки и свойства представляемого ею предмета (понятия), а только те, которые существенны для разрабатываемой программной системы. Тем самым модель «беднее», а следовательно, проще представляемого ею предмета (понятия). Но главное даже не в этом, а в том, что модель есть формальная конструкция: формальный характер моделей позволяет определить формальные зависимости между

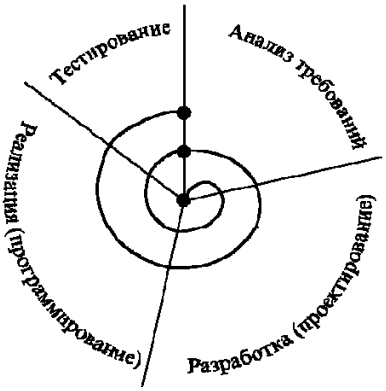

Рис. 1.1. Жизненный цикл программной системы

ними и формальные операции над ними. Это упрощает как разработку и изучение (анализ) моделей, так и их реализацию на компьютере. В частности, формальный характер моделей позволяет получить формальную модель разрабатываемой программной системы как композицию формальных моделей её компонентов.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надёжные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования. Объектно-ориентированный подход является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

Разработка приложений с архитектурой WinRT является на сегодняшний день одной из самых современных и эффективных реализаций данного подхода.

Объектно-ориентированный подход применяется на всех этапах жизненного цикла прикладной информационной системы (рис. 1.1) вне зависимости от базовой архитектуры приложения, начиная с анализа требований к программной системе и её предварительного проектирования, и кончая её реализацией, тестированием и последующим сопровождением.

Жизненный цикл ПО – период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл – процесс построения и развития ПО, который представляет собой набор формализованных процессов [ГОСТ Р ИСО/МЭК 12207–99]:

а) процессы соглашения;

б) процессы организационного обеспечения проекта;

в) процессы проекта;

г) технические процессы;

д) процессы реализации программных средств;

е) процессы поддержки программных средств;

ж) процессы повторного применения программных средств.

Необходимо отметить, что как и во многих других отраслях, при разработке программного обеспечения очень много зависит от постановки

задачи, её первоначальной формулировки, расстановки приоритетов при дальнейшей разработке, определении целей.

Впервые в технологии разработки ПО на основные позиции выводится взаимодействие с пользователем, а не, например, количество компонентов в приложении. Это означает, что концепция взаимодействия приложения с пользователем должна описываться на самых ранних стадиях разработки приложения вплоть до формулирования задания в процессах соглашения. Также при анализе требований к программе на ранних стадиях жизненного цикла необходимо определить ряд ключевых позиций, которые отличают приложения для Windows Store от «традиционных»:

1. Определить содержимое приложения.
2. Определить преимущества приложения.
3. Определить поддерживаемые действия пользователя.
4. Определить поддерживаемые функции.
5. Определить основные концепции интерфейса.

Из вышесказанного следуют три основных отличия в создании приложений для магазина Windows:

1. Анализ требований должен быть направлен на содержание приложения и его удобство с точки зрения пользователя.

2. При проектировании приложений следует сконцентрироваться в первую очередь на данных и пользовательском интерфейсе.

3. Реализация приложения должна основываться на программном интерфейсе WinRT.

Остальные этапы жизненного цикла приложений WinRT являются традиционными. Процессы и задачи на каждом из этапов определяются только уровнем реализации технического задания и будут рассмотрены ниже.

WinRT представляет собой программный интерфейс построения приложений для Windows Store. Интерфейс польностью объектно-ориентированный и поддерживает следующие наборы классов.

Как видно из рис. 1.2, в Windows Runtime выделяют несколько основных блоков.

Базовый набор классов – тут собрано всё по работе с потоками, управлением памятью, ресурсами и аутентификацией приложения. Сюда же можно отнести и определение базовых типов.

Работа с файлами мультимедиа – тут представлены механизмы по работе с аудио и видео.

Службы и данные – сюда входят классы, обеспечивающие возможность взаимодействия с удалёнными службами, а также классы по обработке данных.

Устройство – тут присутствует набор типов, позволяющих взаимодействовать с устройством пользователя, включая различные сенсоры, такие как акселерометр.

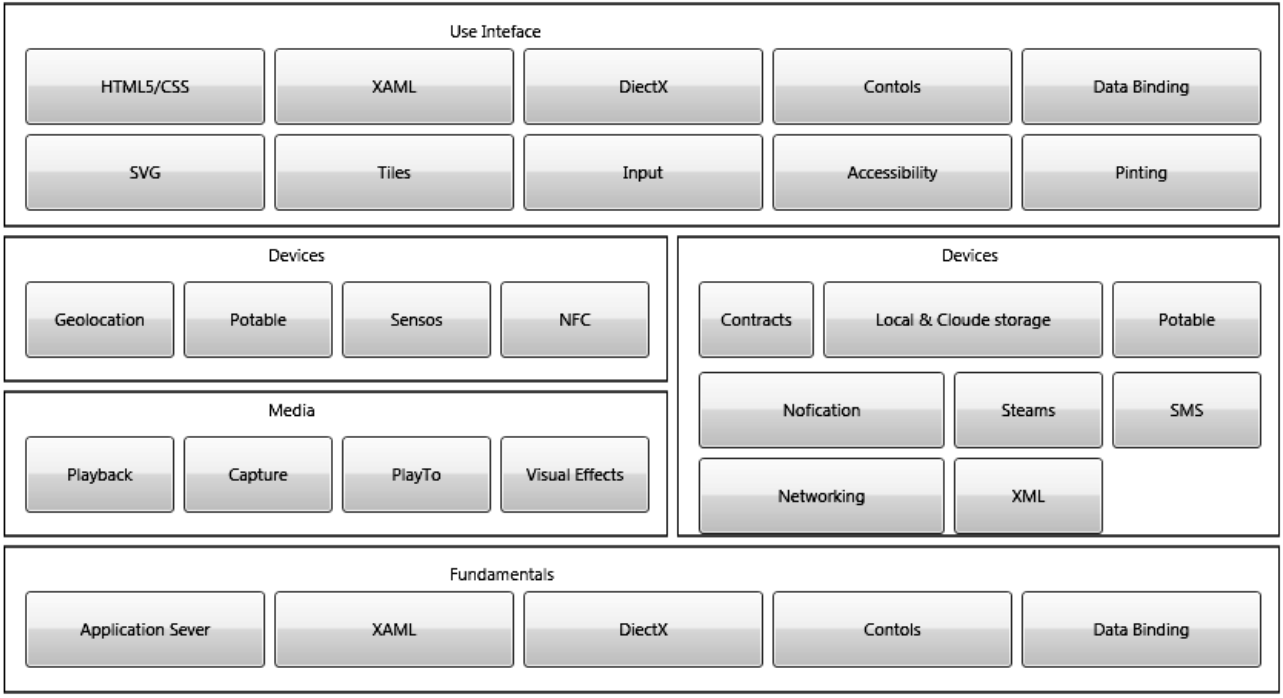

Рис. 1.2. Набор классов WinRT

Пользовательский интерфейс – самый интересный блок, который содержит основные компоненты построения интерфейсов.

Интерфейс WinRT поддерживается различными языками программирования: C++, C#, VisBasic, XAML. Классы WinRT практически во всём совпадают с классами .NET Framework. Это не случайность. Дело в том, что WinRT разрабатывался таким образом, чтобы с его помощью можно было удобно создавать приложения не только на C++, но и на C#. Фактически он адаптировался для C#-разработчиков, и на WinRT можно смотреть как на воплощение .NET Framework в ядре платформы.

Вызов интерфейсов Windows Runtime становится возможным благодаря специальной прослойке Language Projection. Она представляет собой механизм взаимодействия между WinRT и C#/XAML. Проще говоря, под Language Projection можно понимать библиотеку прокси-классов, а также расширения компилятора, делающие разработку приложений на C# «нативной». Подобные прослойки есть и для других технологий. Например, для приложений, написанных на JavaScript/HTML5, существует аналогичная прослойка, которая включает библиотеку WinJS, позволяющую обращаться к WinRT из кода на JavaScript.

2. ИНТЕРФЕЙС ПРИЛОЖЕНИЙ ДЛЯ WINDOWS STORE. ТРЕБОВАНИЯ К ВНЕШНЕМУ ВИДУ ПРИЛОЖЕНИЯ, ИНТЕРФЕЙС ПРИЛОЖЕНИЯ, ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Приложение для магазина Windows разрабатывается на основе технического задания (пусть даже выданного самому себе), которое определяет функционал приложения, и в том числе его интерфейс. Интерфейс приложения для Windows Store состоит из нескольких частей:

- плитка с уведомлениями;
- экран-заставка;
- первый запуск;
- домашняя страница.

Каждая из частей интерфейса имеет собственное функциональное назначение и используется независимо от других частей.

Плитка – это «лицо» приложения. Фактически – это «ярлык на рабочем столе» в терминах предшествующих версий Windows. Каждая плитка выделяет отдельное приложение на рабочем столе. На плитке рекомендуется отражать фирменную символику и подчёркивать ценность приложения. Предусмотрена возможность представлять уведомления на плитке, чтобы приложение всегда вызывало ощущение новизны и актуальности, вновь и вновь привлекая пользователей.

Экран-заставка – используется при инициализации приложения. Должен загружаться как можно быстрее и оставаться на экране только на время, необходимое для инициализации. В экране-заставке рекомендуется отражать идею приложения.

Первые три части интерфейса представляют собой своеобразный глянцевый журнал. Он используется для того, чтобы пользователь заинтересовался приложением, предоставляемым этим приложением услугой и т.п.

Домашняя страница – первая страница приложения, т.е. это место, куда пользователь попадает при каждом запуске приложения. Содержимое домашней страницы должно сразу ясно показывать, для чего предназначено приложение. При разработке страницы очень важно сделать акцент на самом существенном – на данных и их представлении для пользователя. Об остальных возможностях приложения пользователи узнают сами в процессе работы.

Правильно разработанный и реализованный интерфейс приложения должен убедить пользователя в полезности приложения, в его качестве. Наиболее распространёнными ошибками являются требования к пользователю сразу ввести какие-либо личные данные (например создать учётную запись), начать заполнение приложения данными. Правильнее представить пользователю образец содержимого, представить краткий урок по использованию приложения (особенно актуально для сложных систем, типа библиотечной информационный системы), предоставить возможность поэкспериментировать с данными. И только после этого предоставлять пользователю полный функционал приложения.

Части интерфейса реализуются в виде статических элементов и элементов управления. Каждая часть интерфейса, в том числе плитка, может быть реализована как в виде статического компонента, так и динамического. Для этого используется широкая палитра различных компонентов в среде разработки.

Для реализации интерфейса, а также содержания и функционала приложения разработчиком предлагается четыре базовых языка: XAML, Visual Basic, C++ и C#. В данном курсе будут рассматриваться примеры на языке XAML и C++.

Независимо от того, используется ли C# или C++ для разработки Windows 8 приложений, XAML может использоваться совместно с любым кодом. С его помощью в приложении описывается большинство интерфейсных элементов, стилей и ресурсов.

XAML (eXtensible Application Markup Language) представляет собой декларативный язык, построенный на базе XML. Основное назначение этого языка состоит в описании интерфейса приложения XAML.

Чтобы лучше понять назначение XAML, достаточно сравнить создание интерфейса на языке программирования C# или C++ (коды будут очень похожи) при использовании Windows Forms.

Так, часть кода C#, создающего небольшую кнопку, выглядит следующим образом:

```
this.myButton = new System.Windows.Forms.Button();
this.myButton.Location = new System.Drawing.Point(120, 60);
```

```
this.myButton.Name = "myButton";
this.myButton.Size = new System.Drawing.Size(110, 40);
this.myButton.Text = "Hello";
this.Controls.Add(this.myButton);
```

С одной стороны, приведённый выше код является громоздким, а с другой – он не даёт понимания того, как устроен интерфейс. Очень сложно проследить зависимости между контейнерами и понять структуру интерфейса. Это связано с тем, что разобрать и обработать код, написанный на C#, достаточно сложно. Кроме того, код может содержать вставки, не связанные с построением интерфейса.

Рассмотрим аналогичный код на XAML:

```
<Canvas x:Name="LayoutRoot">
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
Width="110" Height="40" x:Name="myButton"></Button>
</Canvas>
```

Как видно, XAML является более интуитивным, чем код на C# или на C++, не только для разработчиков, но и для других членов команды, например дизайнеров. Это позволяет использовать XAML для создания прототипов интерфейсов и переводить их на стадию разработки, используя один и тот же код. По коду XAML легко определить все параметры нашей кнопки и её положение в общей иерархии интерфейсных объектов, а также очень легко построить дерево объектов.

Систаксис языка XAML достаточно подробно рассмотрен в MSDN, а также я ряде других источников и в данном курсе не рассматривается.

Рассмотрим пример построения дизайна и структуры простейшего приложения – информационного справочника о вузе (рис. 2.1). Подробно пример рассматривается в практической части. Остановится на основных моментах.

Первоначально необходимо определить внешний вид данного приложения.

Рис. 2.1. Внешний вид приложения

Создадим разметку и зададим управляющие действия для кнопок. Рассмотрим пример создания данного приложения на основе нескольких примеров и основных функций.

Создание границ для заголовков и меню.

```
<Border x:Name="TopBorder" BorderBrush="#FF08519E" BorderThickness="2" HorizontalAlignment="Left" Height="3" Margin="0,150,0,0" VerticalAlignment="Top" Width="281"/>
```

Создание кнопок управления.

```
<Button x:Name="HeadMasters" Content="Ректор ТГТУ" HorizontalAlignment="Left" VerticalAlignment="Top" Width="353" Foreground="#FF08519E" BorderBrush="#FF08519E" FontFamily="Assets/Play-Regular.ttf#Play" FontSize="20" BorderThickness="0" HorizontalContentAlignment="Left" Background="{x:Null}" IsRightTapEnabled="False" IsTapEnabled="False" IsHoldingEnabled="False" IsDoubleTapEnabled="False" Margin="0,150,0,0" PointerEntered="HeadMasters_PointerEntered" PointerExited="HeadMasters_PointerExited" Click="HeadMasters_Click" />
```

Создание текстовых полей для ввода/вывода информации.

```
<TextBlock x:Name="NotificationBox" HorizontalAlignment="Left" Margin="350,130,0,0" TextWrapping="Wrap" Text="TextBlock" VerticalAlignment="Top" Height="28" Width="600" Foreground="#FF08519E" FontFamily="Assets/Play-Regular.ttf#Play" FontSize="16" Visibility="Collapsed"/>
```

Создание простейшей галереи.

```
<FlipView x:Name="TambovView" HorizontalAlignment="Left" Margin="616,195,0,0" VerticalAlignment="Top" Width="500" Height="333" Visibility="Collapsed">
    <Image Source="Assets/1.jpeg" />
    <Image Source="Assets/2.jpeg" />
    <Image Source="Assets/3.jpeg" />
    <Image Source="Assets/4.jpeg" />
    <Image Source="Assets/5.jpeg" />
</FlipView>
```

Теперь рассмотрим непосредственно процесс хранения и загрузки заметок. В качестве базы данных будет использоваться xml-файл, доступ к которому будет осуществляться с использованием библиотеки Linq. В xml-файле будет один основной элемент noties – так называемый Root-элемент. Внутри него будут располагаться элементы messadges с двумя основными свойствами – name (будет отображать принадлежность к конкретной странице) и value (будет содержать в себе текст заметки).

При этом, когда пользователь нажмёт на соответствующую кнопку – должна произойти загрузка содержимого заметки в текстовое поле.

Рассмотрим функцию, которая получает и записывает в текстовое поле значение заметки.

```
private void searchText(string name_of_search)
{
    ResultBox.Text = "";
    ResultBox.SelectionStart = 0;
```

```
        string fileName = "Assets/Data.xml";
        XDocument doc = XDocument.Load(fileName);
        IEnumerable<XElement> content = doc.Root.Elements("messadge").Where(t
=> t.Attribute("name").Value == name_of_search).ToList();
        string result = "";
        foreach (XElement values in content)
          result += values.Attribute("value").Value.ToString();
        List<string> newList = null;
        char separator = '|';
        newList = result.Split(separator).ToList();
        for (int i = 0; i < newList.Count; i++)
          ResultBox.Text += "      " + newList[i] + Environment.NewLine + Environ-
ment.NewLine;
      }
```

Рассмотрим данную функцию подробнее:

1. name_of_search – содержит в себе ключевое слово для заметки. То есть передача в качестве параметра функции значения “management.rektor” позволит загрузить содержимое заметки о ректоре ТГТУ.

2. ResultBox.Text = ""; – обнуление текстового поля.

3. ResultBox.SelectionStart = 0; – указываем первоначальное положение курсора и полос прокрутки.

4. string fileName = "Assets/Data.xml"; – определяем имя файла, содержащего данные.

5. XDocument doc = XDocument.Load(fileName); – загрузка текущего документа.

6. IEnumerable<XElement> content = doc.Root.Elements("messadge"). Where(t => t.Attribute("name").Value == name_of_search).ToList(); – основная строка в данном приложении. С её использованием осуществляется поиск заметки, атрибут name которой совпадает со значением, переданным пользователем. Затем происходит формирование списка со всеми возможными результатами поиска.

7. foreach (XElement values in content) result += values.Attribute ("value").Value.ToString(); – осуществляется проход по всем существующим (найденным) элементам, полученным в результате поиска, и получение значения заметки.

8. Оставшаяся часть производит разбиение строки в соответствии с разделителями и вывод на экран.

3. ПРОЦЕСС ПРОЕКТИРОВАНИЯ ПРИЛОЖЕНИЙ ДЛЯ WINDOWS STORE

Процесс проектирования приложений заключается в определении внутренних свойств системы и детализации её внешних (видимых) свойств на основе выданных заказчиком требований к приложению. Все требования к приложению указываются в техническом задании на разра-

ботку ПО. Как уже отмечалось выше, от качества реализации технического задания зависит качество всего приложения.

При проектировании приложений для Windows Store программная система представляется в виде трёх взаимосвязанных моделей:

1) объектной модели, которая представляет статические, структурные аспекты системы, в основном связанные с данными;

2) динамической модели, которая описывает работу отдельных частей системы;

3) функциональной модели, в которой рассматривается взаимодействие отдельных частей системы (как по данным, так и по управлению) в процессе её работы, в том числе интерфейс.

Эти три вида моделей позволяют получить три «взаимноортогональных» представления системы в одной системе обозначений. Совокупность моделей системы может быть проинтерпретирована на компьютере (с помощью инструментального программного обеспечения), что позволяет продемонстрировать заказчику (будущему пользователю, тестеру) характер работы с будущей системой и существенно упрощает согласование предварительного проекта системы.

При разработке эскизного проекта WinRT-программа первоначально рассматривается как чёрный ящик. Ход процесса проектирования и его результаты зависят не только от состава требований, но и выбранной модели процесса, опыта проектировщика. Модель предметной области накладывает ограничения на логику приложения, структуры данных и интерфейс.

В российской практике проектирование ведётся поэтапно в соответствии со стадиями, регламентированными ГОСТ 2.103–68: Техническое задание, Техническое предложение, Эскизный проект, Технический проект, Рабочий проект. На каждом из этапов формируется свой комплект документов, называемый проектом (проектной документацией). В зарубежной практике регламентирующими документами, например, являются Software Architecture Document, Software Design Document. При разработке приложений для Windows Store накладываются дополнительные требования к архитектуре ПО, компонентам и интерфейсу.

На этапе эскизного проектирования осуществляется разработка предварительных проектных решений как по приложению в целом, так и по частям. На данном этапе осуществляется планирование приложения. Основываясь на техническом задании, разработчик должен ответить на целый ряд вопросов, таких как:

1. Для чего нужно приложение?
2. Что особенного в приложении?
3. Чего сможет достичь пользователь благодаря приложению?
4. Определить набор функций, реализуемых в приложении.
5. Определить возможности персонализации.

6. Как организовать содержимое пользовательского интерфейса (навигация)?

7. Как организовать команды?

Результатом эскизного проекта является прототип приложения. Прежде чем погрузиться в детальную разработку (рабочий проект), необходимо проверить проект или прототип на соответствие требованиям, нормативам и ожиданиям пользователей во избежание необходимости впоследствии всё переделывать. В рассматриваемой предметной области существуют нормы качества и рекомендации, соблюдение которых поможет значительно улучшить качество приложения. Кроме того, существуют средства для оценки качества продукта с точки зрения разработчика, такие как комплект сертификации приложений для Windows.

Оценка качества прототипа осуществляется в два этапа:

Этап 1: Собственная оценка.

Этап 2: Когнитивный анализ.

Этап собственной оценки – это первый уровень оценки восприятия разрабатываемого приложения пользователями, который основан на поставленных ранее целях. Цель этого метода оценки – обеспечить разработку проекта в соответствии с поставленными задачами. Время, затрачиваемое на этап, – индивидуально для каждого приложения и зависит от количества основных сценариев в нём. Метод оценки можно использовать как при эскизном проектировании, так и в любой другой момент разработки.

К оценке привлекается один или несколько проектировщиков или разработчиков.

Суть метода: перечисление основных механизмов или задач, которые в соответствии с замыслом должны быть предоставлены пользователям в создаваемом приложении. Например, в приложении «Составь слово» задачей может быть набор и отправка одного слова.

В полученном списке задач расставляются приоритеты: от наиболее важной задачи до наименее важной. Выполняются задачи из списка в приложении. Выполняя задачи, сравниваются полученные результаты с шаблоном планирования, который создан в процессе постановки целей на самых первых этапах проектирования. Оценивается степень решения первоначальных задач. Если первоначальные задачи не решены, то в каком состоянии находится приложение? Что нужно сделать, чтобы обеспечить достижение целей?

Когнитивный анализ – это метод оценки, в рамках которого пользователи выполняют в приложении определённые задачи и предоставляют обратную связь от приложения к разработчику в процессе выполнения. Этот метод имеет явное преимущество перед методом собственной оценки, так как обратная связь предоставляется реальными пользователями приложения. Важнейшим аспектом метода является то, что по ходу вы-

полнения поставленных задач пользователи могут выступать с любыми замечаниями и предложениями.

Время, затрачиваемое на анализ, индивидуально для каждого приложения и зависит от количества основных сценариев в нём. Метод используется на любом этапе разработки.

К оценке привлекаются один или несколько пользователей приложения, принадлежащих к целевой аудитории. При наличии конкретного заказчика-потребителя целевая аудитория – это сотрудники заказчика. Если программная система направлена на широкую аудиторию, например для реализации через Windows Store, то необходимо определить целевую аудиторию. Для этого необходимо поставить перед собой ряд вопросов и ответить на них, например:

Кто будет использовать моё приложение?

Каков возраст аудитории?

Что отличает эту аудиторию от других?

При реализации этапа, как и в случае собственного анализа, осуществляется перечисление основных механизмов или задач, которые в соответствии с замыслом должны быть предоставлены пользователям. Задачи выстраиваются в соответствии с приоритетами каждым пользователем в отдельности. Пока пользователи оценивают приложение, осуществите собственную оценку. Поставьте перед собой ряд вопросов, например:

Достигнуты ли в приложении цели, которые я первоначально запланировал?

Справляются ли пользователи с каждой задачей?

С какими проблемами они сталкиваются при выполнении задачи?

Соответствует ли их опыт выполнения задач целям, которые я изначально поставил для своего приложения?

В каком состоянии находится моё приложение? Что нужно сделать, чтобы обеспечить достижение целей?

Результаты анализа всегда следует документировать, делать выводы, выставлять оценки.

Понятие интерфейса

Как отмечалось выше, интерфейс Metro является одним из основных отличий приложений для магазина Windows. Ошибочно думать, что интерфейс Metro – это всего лишь представление приложения в виде «живой плитки» вместо старых иконок на рабочем столе. На самом деле стиль Metro включает много концепций и принципов организации интерфейса приложения.

Концепции и принципы реализуются при проектировании. Рассмотрим основные отличия в проектировании Metro-интерфейсов от «старых» типов приложений.

Старые типы приложений могли содержать не только рамку окна, но и дополнительные панели инструментов, меню, строки состояния и т.д.

Все эти элементы казались неотъемлемой частью любого интерфейса. Очень часто эти элементы занимали достаточно много места на экране и в большинстве случаев не подходили для работы с жестами и касаниями.

Использование стиля Metro предполагает, что пользователь сосредоточен лишь на работе с контентом. Все подобные приложения ориентированы на работу в полноэкранном режиме, а дополнительные элементы управления появляются лишь тогда, когда действительно необходимы.

Среди стандартных элементов можно выделить панель приложения **Application Bar**, которая появляется при нажатии правой кнопки мыши или при выполнении жеста, направленного от нижней границы планшета к центру. Примером реализации такой панели может служить приложение Internet Explorer 10 для Windows Store. Здесь присутствуют сразу две панели приложения, обеспечивающие взаимодействие пользователя с окнами и предоставляющие механизм навигации, но только тогда, когда это необходимо пользователю. В остальное время пользователь работает только с контентом страницы.

Следующий принцип, на котором основывается интерфейс Metro, – это отзывчивость при взаимодействии пользователя с интерфейсом. Это достигается путём вставки анимации, которая запускается при взаимодействии пользователя с тем или иным элементом интерфейса (контентом). Важно то, что пользователь может взаимодействовать с приложением с помощью как мыши, так и пальца. Если взаимодействие осуществляется с помощью мыши, то статичность интерфейса ещё допустима (по крайней мере, курсор мыши движется). Но в случае взаимодействия с интерфейсом с помощью пальцев может возникать ощущение «подвисшего» приложения для статического интерфейса. Ведь не любое взаимодействие с контентом приводит к каким-то конкретным действиям (переход на другую страницу, прокрутка и т.д.). Поэтому необходимо стараться сделать интерфейс динамичным и отзывчивым по отношению к действиям пользователя.

С другой стороны, не следует излишне перегружать анимацией свой интерфейс, необходимо стараться сохранять баланс.

Многие устройства, поступающие сегодня на рынок, поддерживают большой набор разрешений и различные варианты ориентации. Именно поэтому необходимо разрабатывать интерфейс таким образом, чтобы он хорошо работал в любых типах разрешений экрана и с поддержкой альбомной и книжной ориентации.

При работе над несколькими приложениями для Windows 8 удалось увидеть много примеров того, что приложение может успешно работать с мышью, но быть совершенно неприспособленным для работы с жестами. В качестве примера можно взять приложение, отображающее на экран журнал или книгу. Возникает желание реализовать перелистывание страниц так, как это происходит в обычной книге. Пользователь берёт мышь, тянет за край страницы, и она успешно переворачивается. Но как только

пользователь начинает работать с жестами, вместо переворота страниц у него появляется панель управления системы или приложение переключается на другое. Всё дело в том, что система также реализует свои действия в ответ на жесты, и это необходимо учитывать.

В целом, создавая приложение для Windows Store, нужно помнить о следующих вещах:

– работа с помощью касаний и жестов будет преобладать среди пользователей приложения. Поэтому необходимо тщательно тестировать приложение;

– реализовать механизмы взаимодействия с контентом следует таким образом, чтобы они не пересекались со стандартными механизмами, заложенными в операционной системе.

Контракты

Предположим, что разрабатывается специализированное приложение, которое позволяет создавать изображения. Создав изображение, необходимо иметь возможность опубликовать его в Facebook, Twitter и других социальных сетях. Чтобы это сделать, можно реализовать функциональность клиентов Facebook, Twitter и т.д., что не очень правильно, так как потребует изрядных усилий, и подобная функциональность уже присутствует на машине у пользователя.

Кроме того, социальных сетей много, и заранее неизвестно, в какой из них пользователь будет публиковать свои данные. Поэтому логично было бы задействовать уже существующие приложения, предоставив всем программам по работе с социальными сетями специализированный источник данных приложения.

Естественно, чтобы одно приложение могло предоставлять данные, а второе – использовать эти данные, необходимо, чтобы они работали по заранее сформированным правилам или контрактам. И такие контракты есть, Windows 8 предоставляет их в большом количестве.

Плитки и оповещения внутри них не только придают уникальность приложению и создают первое впечатление у пользователя, но и позволяют пользователю создать его собственный уникальный рабочий стол. Именно поэтому уделяйте плиткам как можно больше внимания и попытайтесь определить, можно ли что-то новое показать пользователю (последнюю фотографию, количество непрочитанных сообщений и т.д.). При этом существует два режима отображения плиток: стандартный и расширенный. Поэтому если в какой-то момент информация будет неинтересна пользователю, он просто уменьшит размер плитки приложения.

Обратите внимание на то, что плитки на основном экране можно создавать и из самого приложения. Это необходимо в том случае, когда пользователь хочет определить быстрый механизм доступа к какой-либо из частей приложения. В этом случае наряду с основной плиткой на экране может располагаться и второстепенная плитка. Естественно, разработ-

чик должен позаботиться о том, чтобы предоставить пользователю возможность создать второстепенную плитку.

Сегодня многие устройства оснащены не только передатчиками WiFi, но и 3G-модемами и большую часть времени находятся в сети. Несмотря на это, при проектировании собственного приложения необходимо учитывать и возможность работы после разъединения. Иными словами, приложение должно работать даже при отключении от сети.

В современном мире пользователь может иметь более чем одно устройство. Задача разработчика при проектировании приложений для Windows Store состоит в том, чтобы сделать использование приложения «родным» на любом из устройств. Для этого Windows 8 предлагает множество механизмов, которые позволяют сохранить настройки в «облаке» и использовать их на любой другой машине пользователя.

4. РАЗРАБОТКА ГИБРИДНЫХ ПОДСИСТЕМ. ИСПОЛЬЗОВАНИЕ НЕСКОЛЬКИХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ В ОДНОМ ПРИЛОЖЕНИИ

На сегодняшний день существует большое количество языков программирования для среды Windows. Каждый из языков имеет свои преимущества и недостатки. Приложение, в котором используется несколько языков одновременно, – гибридное, может обладать всеми преимуществами используемых языков. Например, производительность C++ и пользовательский интерфейс XAML.

В Windows 8 имеется возможность создания компонентов на C++, используемых приложением для Windows Store. Такой подход имеет ряд преимуществ:

1. Используется производительность C++ в сложных операциях или операциях с большим объёмом вычислений.
2. Повторное использование существующего кода, который уже написан и протестирован.

В среде Windows 8 можно создавать компоненты (фактически DLL) на языках C++, C# и Visual Basic, а затем использовать полученные компоненты в программах для Windows Store, разработанных на языке JavaScript или XAML. При этом имеется возможность в JavaScript обращаться к API-функциям Windows. Фактически в Windows 8 реализован механизм, который позволяет выбрать язык для создания оболочки определённой функции, и реализовать интерфейс пользователя для этой функции на JavaScript.

При построении решения, содержащего JavaScript и компонент для Windows, файлы проекта JavaScript и скомпилированная библиотека DLL объединяются в один пакет, который можно отлаживать локально, в имитаторе, или удалённо в связанном устройстве. Также можно распространять только проект компонента как расширение SDK.

При написании кода компонента на C++, как правило, используются обычная библиотека C++ и встроенные типы, за исключением абстрактного двоичного интерфейса (ABI), в котором передаются данные в/из JavaScript. Здесь используются типы Windows и специальный синтаксис, поддерживаемый средой Visual C++ для создания этих типов и управления ими. Кроме того, код Visual C++ будет использовать такие типы, как **delegate** и **event**, для реализации событий, которые можно инициировать из компонента и обрабатывать на языке JavaScript.

Рассмотрим основные операции при создании гибридных приложений.

Создание экземпляра объекта

Через интерфейс ABI можно передавать только типы среды Windows. Если компонент содержит такой тип, как **std::wstring**, в качестве возвращаемого типа или параметра открытого метода, компилятор создаёт ошибку. Если используются встроенные типы C++, такие как **int**, **double** и т.д., компилятор автоматически преобразует их в соответствующий тип **int32** среды исполнения, **float64** и т.д. в параметрах и возвращаемых типах открытых методов. Преобразование выполняется только при передаче типа через ABI.

```
namespace MyComp
{
   // объявление класса C++
   public ref class LangSample sealed
   {
      // члены класса
   };
}

//использование в JavaScript
var nativeObject = new myComp.LangSample();
```

Встроенные типы C++, типы библиотек и типы среды Windows.

Экземпляр активируемого класса может быть создан в другом языке, например JavaScript. Для использования такого компонента, например написанного на JavaScript, необходимо наличие в компоненте, по крайней мере, одного активируемого класса.

Компоненты исполняемой среды могут содержать несколько активируемых классов, а также дополнительные классы, которые доступны только для внутреннего использования в компоненте. К типам C++, которые не предназначены для использования на JavaScript, необходимо применить атрибут WebHostHidden.

Активируемый класс должен быть объявлен как public ref class sealed. Ключевые слова класса **ref** указывают компилятору создать класс как тип, совместимый с исполняемой средой, а ключевое слово **sealed** запрещает наследование от этого класса. Для использования в JavaScript класс должен иметь модификатор **sealed**.

Все числовые примитивы определяются в пространстве имён по умолчанию. Platform Namespace – это пространство имён, в котором C++ определяет классы, являющиеся типами среды исполнения Windows. К ним относятся классы: Platform::String и Platform::Object. Конкретные типы коллекций, такие как Platform::Collections::Map и Platform::Collections::Vector, определяются в пространстве имён Platform::Collections. Открытые интерфейсы, реализуемые этими типами, определяются в пространстве имён Windows::Foundation::Collections.

Обмен данными между классами и через ABI осуществляется через стандартные для C++ методы. Некоторые из них рассмотрены ниже.

Метод, возвращающий значение встроенного типа:

```
double LogCalc(double input)
{
   // Use C++ standard library as usual.
   return std::log(input);
}
//Вызов метода
var num = nativeObject.logCalc(21.5);
document.getElementById('callmethod').innerHTML = num;
```

Для передачи определяемых пользователем структур через интерфейс ABI необходимо определить объект JavaScript, который содержит те же члены, что и структура, определённая в C++. Затем можно передать этот объект в качестве аргумента методу C++, чтобы объект был неявно преобразован в тип C++. Другой способ состоит в определении класса, который реализует интерфейс **IPropertySet** (не показан).

```
namespace Test
{   //пользовательская структура
   public value struct BatterData
   {
      Platform::String^ Name;
      int Number;
      double BattingAverage;
   };
   public ref class Test sealed
   {
   private:
      BatterData m_batter;
   public:
      property BatterData Batter
      {
         BatterData get(){ return m_batter; }
      }
   };
}//вызов метода в JavaScript
var myData = nativeObject.batter;
```

document.getElementById('myDataResult').innerHTML = myData.name + " , " + myData.number + " , " + myData.battingAverage.toPrecision(3);

Перегруженные методы

В JavaScript имеется ограниченная возможность различения перегруженных методов. Например, JavaScript может определить различия между следующими сигнатурами:

int GetNumber(int i);
int GetNumber(int i, string str);
double GetNumber(int i, MyData^ d);

Однако между следующими сигнатурами язык не определит разчилий:

int GetNumber(int i);
double GetNumber(double d);
или этими сигнатурами такого вида:
MyData^ GetData(MyData^ d1, MyData^ d2);
MyData^ GetData(MyData^ d1, TestData^ d2);

В случаях неоднозначности можно добиться, чтобы код JavaScript всегда вызывал конкретную перегрузку путём применения атрибута Windows::Metadata::DefaultOverload к сигнатуре метода в файле заголовка.

//C++ header file:
[Windows::Foundation::Metadata::DefaultOverloadAttribute]
int WinRTComponent::GetNumber(int i);

double WinRTComponent::GetNumber(double d);

Этот код JavaScript всегда вызывает перегрузку с атрибутом:

var num = nativeObject.getNumber(9);

Коллекции и массивы

Коллекции всегда передаются через интерфейс ABI в качестве дескрипторов типов среды исполнения Windows, таких как Windows::Foundation::Collections::IVector^ и Windows::Foundation:: Collections::IMap^. Например, если возвращается дескриптор типа Platform::Collections::Map, он будет неявно преобразован в Windows::Foundation::Collections::IMap^. Интерфейсы коллекций определяются в отдельном пространстве имён, состоящем из классов C++, которые предоставляют конкретные реализации.

Свойства

Открытые элементы данных, такие как свойства, необходимо представлять с помощью ключевого слова property. Тривиальное свойство аналогично элементу данных, поскольку вся его функциональность является неявной. Нетривиальное свойство имеет явные методы доступа get и set и закрытую переменную с именем, которая является "резервным хранилищем" для значения.

Делегаты и события

delegate – это тип среды исполнения Windows, представляющий объект функции. Делегаты можно использовать в связи с событиями, обрат-

ными вызовами и асинхронными вызовами методов, чтобы задать действие, которое будет выполнено позже. Подобно объекту функции, делегат обеспечивает безопасность типа, позволяя компилятору проверять тип возвращаемого значения и типы параметров функции. Объявление делегата напоминает сигнатуру функции, реализация аналогична определению класса, а его вызов похож на вызов функции. Экземпляр делегата может быть также создан "встроенным" с помощью лямбда-выражения.

Асинхронные методы

Чтобы использовать асинхронные методы, предоставляемые другими объектами среда исполнения, используйте класс task. Для реализации асинхронных методов в C++ необходимо использовать функцию Create_async(), которая определена в файле ppltasks.h..

Исключения

Можно создавать исключения любого типа, определённого в среде исполнения Windows. От исключений среды нельзя наследовать пользовательские типы. Однако можно создать исключение COMException и предоставить пользовательский объект HRESULT, который может быть доступен для кода, перехватывающего исключение. Способы задания пользовательского сообщения в исключении COMException не предусмотрены.

Отладка всех компонентов разрабатываемого приложения осуществляется стандартными средствами среды разработки. При отладке решения JavaScript, содержащего библиотеку DLL компонента, можно настроить отладчик для пошагового выполнения скрипта или машинного кода в компоненте, однако нельзя отлаживать эти части одновременно. Чтобы изменить этот параметр, разверните узел проекта JavaScript в обозревателе решений, а затем последовательно выберите пункты Свойства, Отладка, Тип отладчика.

Обязательно установить соответствующие возможности в конструкторе пакетов. Например, если требуется открыть файл с помощью интерфейсов API среды исполнения Windows, необходимо установить флажок Доступ к библиотеке документов в области Возможности конструктора пакетов.

Если коду JavaScript не удаётся распознавать открытые свойства или методы в компоненте, убедитесь, что в JavaScript используется "верблюжий" стиль имён. Например, метод LogCalc C++ следует вызывать из JavaScript как logCalc.

При удалении проекта компонента из среды выполнения Windows C++ из решения, необходимо также вручную удалить ссылку на проект из проекта JavaScript. Невыполнение этого требования приведёт к невозможности последующей отладки и выполнения операций построения. При необходимости можно добавить ссылку на сборку в библиотеку DLL.

5. АРХИТЕКТУРА ПРИЛОЖЕНИЯ ДЛЯ WINDOWS STORE

WinRT была создана для предоставления единообразного (в то же время изменчивого с течением времени в плане новых веяний) и безопасного окружения для работы Windows-приложений. WinRT испытала на себе влияние .NET, C++ и JavaScript. Однако стоит учитывать, что WinRT не заменяет собой CLR или Win32, а предоставляет унифицированную среду выполнения приложений, написанных на разных языках, для запуска в новых версиях Windows с использованием интерфейса Metro.

По сути своей WinRT это новый набор API, основные особенности которого:

– обеспечение пользовательского интерфейса Metro;
– упрощённое программирование GUI;
– WPF и XAML для построения интерфейсов.

Все API-вызовы асинхронны. API изначально разработана как «песочница» (приложения сразу готовы к размещению в WindowsStore). Определения API-вызовов представлены в формате метаданных ECMA 335 (лежат в файлах .winmd) (рис. 5.1).

WinRT призвана уменьшить степень взаимовлияния приложений – производительность одного приложения не должна влиять на производительность другого; ресурсы и объекты одного приложения доступны другим только через стандартные каналы ОС (контракты).

Основная идея, которую воплощают собой контракты, – «Приложения должны иметь возможность тесно работать друг с другом и при этом ничего не знать друг о друге». В этом помогает новый компонент системы – ShareBroker, который является посредником между приложениями – одно приложение готовит буфер (пакет данных с какой-либо информацией, которой собирается поделиться с другими приложениями) и отправляет его ShareBroker'у, вызывающее приложение в свою очередь запрашивает интересующий пакет у ShareBroker'а, а не у приложения-источника.

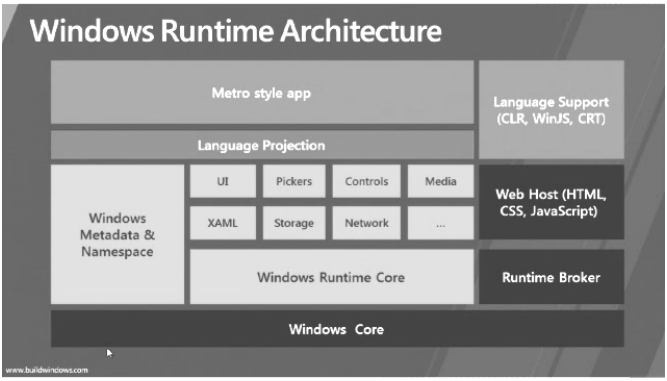

Рис. 5.1. Архитектура WinRT

Вот некоторые из типов контрактов:

– **Search** – благодаря этому контракту приложение может стать источником поиска данных. Поиск внутри вашего приложения появляется на панели инструментов операционной системы, но данным контрактом могут воспользоваться и другие приложения;

– **Sharing** – с помощью этого контракта можно открыть доступ к данным других приложений. Выше был описан пример интеграции приложения с социальными сетями;

– **Play To** – этот контракт предназначен для передачи медиаданных из приложения на внешние устройства;

– **Settings** – данный контракт предоставляет доступ к настройкам приложения, благодаря чему система может интегрировать все настройки в свою панель инструментов;

– **App to App** – благодаря этому типу контрактов приложения могут не только обмениваться данными, но и предоставлять доступ к различным хранилищам, которыми оперируют;

– **Picker** – описывается какими данными приложение хочет поделиться с другими, что делает простым доступ к файлам этого приложения извне, унифицируя интерфейс доступа к виду как будто бы происходит работа с жёстким диском.

Рассмотрим механизм работы контрактов. Для примера возьмём контракт share («Отправка»), состоящий из двух частей share source и share target. Приложение, которое хочет чем-то поделиться с другим приложением, реализует свою часть контракта (share source), принимающее – свою (share target) при этом приложениям ничего не нужно знать друг о друге, а только позаботится об исполнении своей части «контракта», об остальном (приём/передача информации и унифицированность процесса) позаботится ОС.

Share Contract

Очень важно помнить, что данная функциональность работает только для WinRT-приложений и недоступна на традиционном рабочем столе.

Для передачи через share contract можно использовать:

- простой текст;
- URI;
- HTML;
- отформатированный текст;
- изображение;
- файл;
- данные в формате, представленном разработчиком.

Использование контрактов Share настолько широко, что привело к появлению сленгового термина «расшаривать». Поскольку с английского Share переводится по-разному, в зависимости от используемого контекста, то и однозначного значения термина «расшаривать» нет. Наиболее близкое по смыслу слово «поделиться».

Пример с расшариванием текста:

```
// подключаем пространство имён, которое необходимо для базового шаринга
using Windows.ApplicationModel.DataTranfer;
/*в случае если нужно больше используются
Windows.Storage – для объекта StorageFile (передача файлов)
Windows.Storage.Pickers для открытия диалога выбора файлов/изображений
Windows.Storage.Streams часто используется при передаче файлов/изображений/данных в собственном формате
Windows.Graphics.Imaging используется для изменения изображений перед расшариванием (создание предварительного просмотра, уменьшение размеров, поворот на угол, увеличение и т.п.)
*/
```

Чтобы сообщить ОС, что приложение собирается что-то расшарить в функции OnNavigatedTo, следует сделать следующее:

```
// зарегистрируем приложение как источник данных (Share)
this.dataTransferManager = DataTransferManager.GetForCurrentView();
this.dataTransferManager.DataRequested += new TypedEventHandler<DataTransferManager, DataRequestedEventArgs>(this.OnDataRequested);
```

В коде выше DataTransferManger – это поле класса выбранной страницы типа DataTransferManager, а OnDataRequested – callback-функция для обработки запроса на расшаривание.

Функция вызывается при нажатии на charm Sharing. Содержимое функции OnDataRequsted^:

```
private void OnDataRequested(DataTransferManager sender, DataRequestedEventArgs e)
{   string dataPackageTitle = TitleInputBox.Text;
if (!String.IsNullOrEmpty(dataPackageTitle))
    {
        string dataPackageText = TextToShare.Text;
        if (!String.IsNullOrEmpty(dataPackageText))
        {
            DataPackage requestData = e.Request.Data;
            requestData.Properties.Title = dataPackageTitle;
                string dataPackageDescription = DescriptionInputBox.Text;
            if (dataPackageDescription != null)
            {
                requestData.Properties.Description = dataPackageDescription;
            }
            requestData.SetText(dataPackageText);
        }
        Else {/// Нечем делиться
        }
    }
    else    {  // Нет обязательного заголовка пакета
    }
}
```

На этом работа приложения, которое хочет поделиться текстом с другим приложением, заканчивается. Расшаривание другого контента выглядит аналогично и состоит из тех же шагов.

1) сообщить операционной системе, что приложение поддерживает контракт sharing;

2) заполнить служебную информацию о DataPackage (заголовок (обязательно), описание (желательно));

3) заполнить и отправить данные через SetText, SetUri etc для стандартных вариантов и через SetData (будет рассмотренно ниже) для своего формата данных.

Рассмотрим отдельно открытие доступа (расшаривание) к файлам, чтобы описать контракт FilePicker и расшаривание данных своего формата для того, чтобы полностью закончить описания шаринга в Windows 8.

При открытии доступа к файлу порядок действий тот же с одним исключением – пользователю нужно выбрать, какие файлы он хочет передать другому приложению. (Этот случай используется здесь только для демонстрации FilePicker'a в Windows 8 имеется возможность программно инициировать Sharing)

Сначала нужно объявить переменную, в которой будет храниться список файлов (1 или более):

```
private IReadOnlyList<StorageFile> storegeItems;
```

Затем в качестве callback-функции для события Clicked для какой-либо кнопки, например с надписью "Выбрать файлы", реализовать следующее:

```
FileOpenPicker filePicker = new FileOpenPicker
{
   ViewMode = PickerViewMode.List,
   SuggestedStartLocation = PickerLocationId.DocumentsLibrary,
   FileTypeFilter = { "*" }
};

IReadOnlyList<StorageFile> pickedFiles = await filePicker.PickMultipleFilesAsync();

if (pickedFiles.Count > 0)
{
   this.storegeItems = pickedFiles;
}
```

Теперь можно подготавливать пакет данных, складывать в него файлы и отправлять:

```
private void OnDataRequested(DataTransferManager sender, DataRequestedEventArgs e)
{
   string dataPackageTitle = "shared file exapmle";
   string dataPackageDescription = "Here we will transfer some file(s) to another app";
```

```
        DataPackage req = e.Request.Data;
        req.Properties.Title = dataPackageTitle;
        req.Properties.Description = dataPackageDescription;
        req.SetStorageItems(this.storegeItems);
    }
```

Расшаривание данных собственного формата. Например, это информация о книге в формате json:

```
    string DataPackageFormat = "book-info";
        string DataPackageText =
        @"{
          ""type"" : ""http://schema.org/Book"",
          ""properties"" :
          {
           ""image"" : ""http://sourceurl.com/catcher-in-the-rye-book-cover.jpg"",
           ""name"" : ""The Catcher in the Rye"",
           ""bookFormat"" : ""http://schema.org/Paperback"",
           ""author"" : ""http://sourceurl.com/author/jd_salinger.html"",
           ""numberOfPages"" : 224,
           ""publisher"" : ""Little, Brown, and Company"",
           ""datePublished"" : ""1991-05-01"",
           ""inLanguage"" : ""English"",
           ""isbn"" : ""0316769487""
          }
        }";
```

А теперь отправим эти данные:

```
            if (!String.IsNullOrEmpty(dataPackageText))
            {
                DataPackage requestData = e.Request.Data;
                requestData.Properties.Title = dataPackageTitle;

                // The description is optional.
                string dataPackageDescription = DescriptionInputBox.Text;
                if (dataPackageDescription != null)
                {
                    requestData.Properties.Description = dataPackageDescription;
                }
                requestData.SetData(dataPackageFormat, dataPackageText);
            }
```

Как видно из кода, единственная разница между своим форматом данных и стандартными заключается в одной единственной строчке:

```
requestData.SetData(dataPackageFormat, dataPackageText);
```

А теперь рассмотрим вторую часть контракта – сторона принимающего приложения. Сначала в манифесте (рис. 5.2) приложения необходимо указать, что оно выступает в качестве share target (реализует эту часть контракта), затем в настройках этого объявления указать, какие типы данных и файлов оно может принимать.

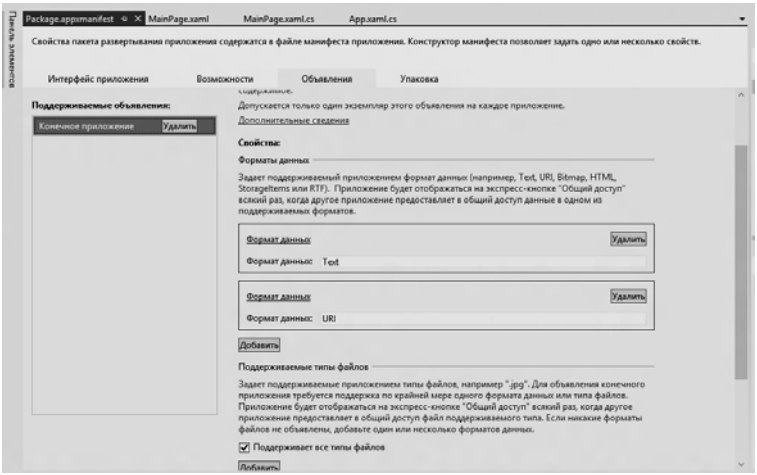

Рис. 5.2. Манифест приложения

Затем в функцию OnNavigatedTo следует добавить код вроде этого:

```
this.shareOperation = (ShareOperation)e.Parameter;
var unused = Task.Factory.StartNew(async () =>
{
  // Получим свойства пакета данных
  this.sharedDataTitle = this.shareOperation.Data.Properties.Title;
  this.sharedDataDescription = this.shareOperation.Data.Properties.Description;
  this.sharedThumbnailStreamRef = this.shareOperation.Data.Properties.Thumbnail;
  this.shareQuickLinkId = this.shareOperation.QuickLinkId;

  // а теперь непосредственно данные
  // проверяем, что именно мы получили и обрабатываем
  If(this.shareOperation.Data.Contains(StandardDataFormats.Text))
  {
      try
      {
         this.sharedText = await this.shareOperation.Data.GetTextAsync();
      }
      catch (Exception ex)
      {
      }
  }
  // И так далее для всех зарегестрированных типов данных
  // однако не стоит забывать про this.shareOperation.Report*()
  // где * -это Started, DataRecieved, Completed для сообщения ОС этапов отработки контракта.
```

Результат испытания показан на рис. 5.3.

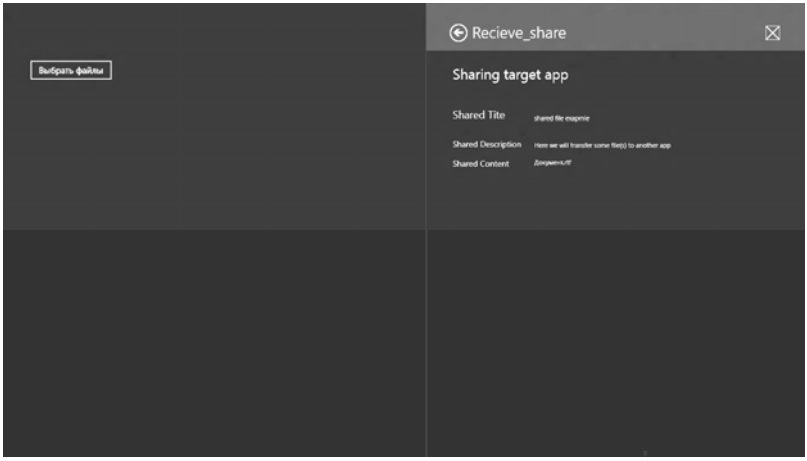

Рис. 5.3. Приём и передачи файла

6. ЖИЗНЕННЫЙ ЦИКЛ ПРИЛОЖЕНИЯ С АРХИТЕКТУРОЙ WINRT. ОСОБЕННОСТИ АРХИТЕКТУРЫ. ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТАРИЯ CLR, .NET. МЕТОДЫ ПРОГРАММИРОВАНИЯ

Поскольку все приложения WinRT ориентируют пользователя на работу с контентом, то нет необходимости держать приложение в памяти, когда пользователь начал работу с другим приложением. С другой стороны, пользователь может в любой момент попытаться вернуться в приложение, поэтому производить его запуск с нуля тоже плохо. В связи с этим Windows 8 приостанавливает работу приложения всякий раз, когда пользователь переключается на какое-то другое занятие, и держит это приложение в памяти, пока есть возможность. Как только пользователь возвращается в приложение, Windows 8 может либо активировать существующий в памяти экземпляр, либо же, если приложение было удалено, вызвать его снова (рис. 6.1). Именно поэтому у разработчика возникает ряд задач.

1. Как только приложение переходит в состояние ожидания, данные, критичные для его возобновления, необходимо сохранить.

2. В случае возобновления работы приложения из памяти необходимо определиться, нужно ли обновлять интерфейс приложения (так как за время его «сна» многое могло измениться) и восстанавливать сетевые подключения. Ведь если приложение оставалось в режиме сна слишком долго, проще обновить весь интерфейс с нуля.

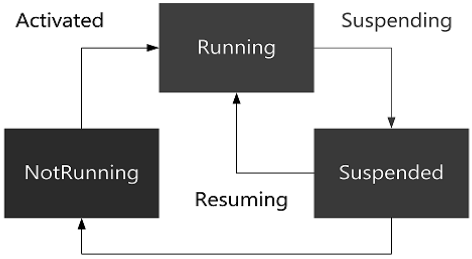

Рис. 6.1. Жизненный цикл приложения WinRT

3. В случае возобновления работы приложения необходимо считать все сохранённые данные и запустить приложение с того состояния (если это возможно), на котором пользователь покинул приложение, либо же перезапустить приложение в штатном режиме.

Приложение может быть приостановлено, если пользователь переключается на другую задачу или если Windows переводит компьютер в режим пониженного энергопотребления. Пока приложение приостановлено, оно продолжает находиться в памяти, поэтому пользователи могут быстро и надёжно переключаться между приостановленными приложениями и возобновлять их работу. Когда приложение приостанавливается, а затем возобновляет работу, не нужно создавать дополнительный код для восстановления первоначального вида приложения.

Однако Windows может в любое время завершить работу приостановленного приложения, чтобы освободить память для других программ или для экономии энергии. Если приложение завершается, оно останавливает свою работу и выгружается из памяти.

Когда пользователь закрывает приложение нажатием клавиш Alt+F4 или жестом закрытия, приложение приостанавливается на 10 секунд и затем завершает работу.

Windows передаёт приложению сообщение о том, что оно будет приостановлено, но дополнительно не предупреждает о завершении работы приложения. Поэтому приложение должно обрабатывать событие приостановки, чтобы сохранить своё состояние и немедленно освободить собственные монопольные ресурсы и дескрипторы файлов.

Для успешного взаимодействия с пользователем необходимо восстанавливать первоначальный вид приложения, который был до приостановки. Это означает, что приложение должно сохранять любые введённые пользователем данные, изменённые им параметры и т.д. Состояние приложения должно сохраняться при приостановке приложения на случай, если Windows завершит его. Только при таких условиях впоследствии можно восстановить состояние приложения.

В общем случае в приложении имеется два типа данных, которыми нужно управлять: данные приложения и данные сеанса.

Чтобы реализовать все вышеописанные механизмы, в классе Application (C#, C++) представлено несколько полезных событий. Давайте их рассмотрим.

Activated – событие генерируется в случае успешного запуска приложения. Теоретически именно здесь необходимо определять, как именно было запущенно приложение, нужно ли восстанавливать состояние и др. На практике переопределить это событие нельзя, и тут существуют другие механизмы.

Suspending – это событие позволяет сохранить текущее состояние приложения. На это у приложения есть 5 секунд, после чего работа будет завершена в любом случае. Ничего сложного в рассматриваемом событии нет, достаточно определить обработчик и использовать локальное хранилище для сохранения данных.

Resuming – событие, которое даёт шанс разработчику восстановить внешние ресурсы в случае, если приложение было запущено из памяти. Очевидно, что во время «сна» приложения сетевые соединения были разрушены, да и контент мог измениться. Тут нужно учитывать, что обработчик этого события не запускается в интерфейсном потоке, следовательно, нужно использовать Dispatcher.

Вернёмся к событию Activated. Дело в том, что приложение может быть запущено по многим причинам: для использования приложения как источника поиска; в результате передачи какого-то файла данных; вследствие активации через FilePicker и др. Именно поэтому, чтобы облегчить жизнь разработчику, класс Application предлагает несколько методов перегрузки. Примерами таких методов могут служить OnFileActivated, OnSearchActivated, OnShareTargetActivated и др. Иными словами, если запуск Вашего приложения связан с инициацией какого-то контракта, то нужно попробовать найти метод, соответствующий этому контракту. Если метод для контракта не найден, то можно воспользоваться методом OnActivated, который получает в качестве параметров тип контракта async protected override void OnActivated(IActivatedEventArgs args).

```
{
switch (args.Kind)
{
case ActivationKind.CameraSettings:
. . . .  break;
case ActivationKind.ContactPicker:
. . . break;
case ActivationKind.PrintTaskSettings:
. . . .  break; . . . .
}
base.OnActivated(args);
}
```

Если же не интересуют контракты, то рекомендуется перегружать метод OnLanched (он уже есть у Вас в коде) и именно тут делать проверки на состояние, из которого было запущено приложение, а также принимать решения, какие данные нужно обновлять и какую страницу делать основной. Параметр метода OnLauched содержит одно полезное свойство – PreviousExecutionState, которое позволяет определить, из какого состояния было запущено приложение.

Для реализации переходов между различными состояниями жизненного цикла необходимо осуществлять хранение данных. У приложения должна быть возможность сохранять данные – настройки, сведения о состоянии и данные для будущей синхронизации. К тому же следует обеспечить комфортную работу пользователя на различных устройствах.

Данные, предназначенные для хранения, могут быть представлены и как набор переменных простых типов, и в виде файлов различной структуры. В связи с этим в Windows Runtime выделяют три способа хранения данных приложения.

– Локальные данные – все настройки сохраняются в реестре или внутри файлов, ассоциированных с конкретным пользователем.

– Роуминг данных – данные размещаются в «облаке» и могут использоваться приложением на различных устройствах.

– Временные данные – данные размещаются во временном хранилище и могут быть удалены в любой момент.

Рассмотрим детально локальное хранение данных и временные данные. Роуминг данных будет рассмотрен далее.

Хранение данных локально.

Существует два локального хранения данных:

1. Хранить простые данные внутри реестра.
2. Хранить данные внутри файлов.

Разумеется, что поскольку приложения WinRT работают внутри собственной «песочницы», то прямого доступа к реестру или файловой системе у них нет. По этой причине Windows Runtime предлагает специальный набор классов, который позволяет хранить данные в выделенном разделе реестра или папке, ассоциированной с конкретным пользователем.

Представленные классы выполняют всю черновую работу. По большому счёту разработчик может и не догадываться, где хранятся данные. Доступны следующие классы.

– ApplicationDataContainer – представляет собой контейнер, куда можно сохранять данные простых типов или данные, построенные на базе.

– ApplicationDataCompositeValue. С каждым приложением ассоциируется контейнер по умолчанию, но можно и принудительно создавать вложенные контейнеры (аналог ключей и папок в реестре).

– ApplicationDataCompositeValue – этот класс позволяет собрать сложный тип данных на основании нескольких простых. Фактически ука-

занный механизм облегчает группировку данных простых типов. Подобное можно сделать и с помощью вложенных контейнеров, но рассматриваемый класс более прост в использовании.

Перечисленные выше классы находятся в пространстве имён Windows.Storage и могут быть использованы без особых затруднений. Например, если необходимо получить контейнер по умолчанию или создать новый именованный контейнер, то можно применить следующий код:

```
ApplicationDataContainer current = ApplicationData.Current.LocalSettings;
ApplicationDataContainer named = current.CreateContainer(
"myContainer", ApplicationDataCreateDisposition.Always);
```

Этот код получает ссылку на основной контейнер, ассоциированный с пользователем приложения, а затем создаёт вложенный контейнер. Имея ссылку на контейнер, можно приступить к сохранению простых данных.

При взаимодействии с реестром поддерживаются следующие типы данных: Boolean, Double, Int32, Int64, Single, String, UInt8, UInt32, UInt64.

Код ниже демонстрирует, как можно добавить новую пару ключ-значение в контейнер.

```
current.Values["myValue"] = 5;
current.Containers["myContainer"].Values["mySecondValue"] = "Hello";
```

В примере присутствуют простые индексаторы, которые позволяют легко создать пару или получить значение по известному ключу.

Наконец, если требуется создать в контейнере комплексное (композитное) значение, то можно воспользоваться следующим кодом.

```
ApplicationDataCompositeValue composite =new ApplicationDataCompositeValue();
composite["fi rstVal"] = 1;
composite["secondVal"] = "Hello";
current.Values["compValue"] = composite;
```

Гораздо интереснее работать с файлами. Для их хранения приложению также выделяется отдельный контейнер, с которым можно взаимодействовать при помощи класса **StorageFolder**.

```
StorageFolder current = ApplicationData.Current.LocalFolder;
```

Созданный выше объект располагает множеством интересных методов.

– CreateFileAsync – создаёт файл.
– CreateFolderAsync – создаёт каталог.
– DeleteAsync – удаляет объект (файл или каталог).
– RenameAsync – позволяет выполнить переименование объекта.
– GetFileAsync – позволяет вернуть ссылку на файл с указанным именем (объект типа StorageFile).
– GetFilesAsync – возвращает список файлов, соответствующих запросу.
– GetFolderAsync – возвращает указанный в параметрах каталог.

– GetFoldersAsync – возвращает список каталогов, соответствующих запросу.

– OpenStreamForReadAsync – открывает файл для чтения, позволяет вернуть объект типа Stream, представленный в платформе .NET Framework уже давно. Этот и следующий метод можно использовать для стандартного .NET взаимодействия с файлами.

– OpenStreamForWriteAsync – аналогичный предыдущему методу, но открывает файл для записи.

– OpenAsync – позволяет открыть файл и обойтись только механизмами Windows Runtime, предоставляя механизмы работы с файлом через интерфейсы.

Рассмотрим небольшой пример, демонстрирующий создание файла в корневом каталоге приложения.

```
public async void WriteFile()
{
StorageFolder current = ApplicationData.Current.LocalFolder;
StorageFile fi le = await current.CreateFileAsync(
"hello.txt", CreationCollisionOption.ReplaceExisting);
IRandomAccessStream writeStream =
await fi le.OpenAsync(FileAccessMode.ReadWrite);
IOutputStream outputStream = writeStream.GetOutputStreamAt(0);
DataWriter dataWriter = new DataWriter(outputStream);
dataWriter.WriteString("hello");
await dataWriter.StoreAsync();
outputStream.FlushAsync();
}
```

Приведённый код не очень простой и требует регулярного обращения к операторам **await**, да и сам метод был объявлен с модификатором **async**. В примере использовались только возможности WinRT, не прибегая к механизмам .NET, а чтобы не активировать механизмы работы с массивом байтов, применили вспомогательный класс **DataWriter**. Этот класс может использовать поток, чтобы облегчить процедуры чтения и записи в файл.

Процедура чтения данных из файла выглядит аналогично.

```
public async void ReadFile()
{
StorageFolder current = ApplicationData.Current.LocalFolder;
StorageFile sampleFile = await current.GetFileAsync("hello.txt");
IRandomAccessStream readStream =
await sampleFile.OpenAsync(FileAccessMode.Read);
IInputStream inputStream = readStream.GetInputStreamAt(0);
DataReader dataReader = new DataReader(inputStream);
string myString = dataReader.ReadString((uint)readStream.Size);
}
```

7. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА METRO. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ. СТИЛИ. РЕСУРСЫ.

Для реализации элементов интерфейса в Winodws 8 используются базовые элементы управления. В среде разработки Visual Studio имеется стандартный набор элементов управления, которые являются неотъемлемой частью любого интерфейса.

Класс Control

Класс является базовым для всех элементов управления, наделяя их базовыми свойствами Windows.UI.Xaml.Controls.Control. Начнём с трёх свойств, которые определяют цветовые характеристики элементов управления:

– Background – тут содержатся свойства фона, который, как правило, является поверхностью элемента управления;

– Foreground – это свойство задаёт цвет текста, который может присутствовать в качестве содержимого элементов управления;

– Opacity – это свойство (наследуется от UIElement) определяет прозрачность элемента управления. Прозрачность задаётся в процентах и варьируется от 0 до 1.

Свойства Background и Foreground могут принимать объект типа Brush, описывающий кисть. Рассмотрим пример работы с самой простой кистью – SolidBrush:

```
<StackPanel>
<Button Background="#FFFF0000" Foreground="#FF00FF00"
Content="Hello"></Button>
<Button Background="Red" Foreground="LightGreen"
Content="Hello"></Button>
<Button Content="Hello">
<Button.Background>
<SolidColorBrush Color="Red"></SolidColorBrush>
</Button.Background>
<Button.Foreground>
<SolidColorBrush Color="LightGreen"></SolidColorBrush>
</Button.Foreground>
</Button>
</StackPanel>
```

В этом примере цвет задаётся сразу тремя способами: в шестнадцатеричном представлении, с использованием возможности конвертера значений свойств в XAML, явным созданием объекта SolidColorBrush. Аналогично свойства можно устанавливать и из C# кода:

```
btn.Background = new SolidColorBrush(Colors.Red);
```

Тут Colors представляет собой специальный класс, определяющий несколько популярных цветов.

Если используемый элемент работает с текстом, то для установки шрифта текста могут пригодиться следующие свойства:

– FontFamily – задаёт имя шрифта, который Вы хотите использовать;

– FontSize – размер шрифта в единицах Silverlight;

– FontStyle – позволяет задать стиль шрифта, такой как Normal или Italic;

– FontWeight – задаёт вес текста, например Bold;

– FontStretch – используется для поддержки шрифтов OpenType, в частности позволяет сжимать или растягивать текст.

Ниже приведён пример использования перечисленных свойств, применяемых для установки надписи кнопки:

```
<Button Background="Red" Foreground="LightGreen" Content="Hello"
FontFamily="Arial"
FontSize="16"
FontStyle="Italic"
FontWeight="Bold">
</Button>
```

Следующие несколько свойств позволяют задать размеры элемента, а также отступы от соседних элементов и от содержимого:

– Width – длина элемента управления;

– Height – ширина элемента управления;

– Padding – расстояние от содержимого до границ элемента;

– Margin – расстояние от каждой из границ до границ соседних элементов.

Кнопки

В WinRT представлено большое количество кнопок. Среди них и новые типы, такие как ToggleSwitch. Вот краткое описание существующих кнопок:

– Button – классическая кнопка, содержащая в качестве контента любой объект типа UIElement и инициирующая событие Click;

– ToggleButton – тип кнопки, используемый для имитации эффекта залипания. Кнопка может быть в двух состояниях: нажатой и отпущенной. Чтобы обрабатывать изменение состояния кнопки, используется событие Checked;

– RepeatButton – в отличие от классической кнопки, этот вид кнопки позволяет непрерывно генерировать событие Click, если кнопка остаётся нажатой, т.е. удерживается пользователем в этом состоянии;

– CheckBox – эта кнопка является прямым наследником ToggleButton и реализует простейший флажок. В отличие от ToggleButton, флажок можно устанавливать в промежуточное состояние, но только из кода и только если свойство IsThreeState установлено как true;

– RadioButton – этот элемент управления полностью аналогичен CheckBox, но предоставляет возможность размещать целый набор элементов в группы, позволяя сделать единственный выбор;

– HyperlinkButton – подобный тип кнопки чаще всего используется внутри текста для организации переходов на другие страницы. Кнопка по умолчанию выглядит как обычная гиперссылка в браузере;

Рис. 7.1

– ToggleSwitch (рис. 7.1) – этот тип кнопки крайне интересен в условиях появления Touch-интерфейса, так как с его помощью можно легко выполнять переключения состояния. Элемент очень напоминает CheckBox, но имеет совершенно уникальное представление.

Рис. 7.2. Группа элементов RadioButton

Поскольку ToggleSwitch (рис. 7.2) является новым элементом управления, хотелось бы продемонстрировать пример его работы. Код ниже задаёт такой элемент и позволяет выполнять переключение между двумя состояниями «Да» и «Нет»:

<ToggleSwitch OffContent="Нет" OnContent="Да"></ToggleSwitch>

Этот элемент пришёл с платформы Windows Phone и полностью оптимизирован для работы с помощью пальцев.

Следующий пример, демонстрирующий создание группы элементов RadioButton:

```
<StackPanel>
<RadioButton Content="Choice 1" IsChecked="True"
GroupName="Group 1" Margin="5">
</RadioButton>
<RadioButton Content="Choice 1" IsChecked="False"
GroupName="Group 1" Margin="5">
</RadioButton>
<RadioButton Content="Choice 1" IsChecked="False"
GroupName="Group 1" Margin="5">
</RadioButton>
<RadioButton Content="Choice 1" IsChecked="False"
GroupName="Group 1" Margin="5">
</RadioButton>
</StackPanel>
```

Как сказано выше, особенность элементов управления в XAML состоит в том, что в качестве содержимого у любого из элементов (за редким исключением) может выступать контейнер или другой элемент. Ниже код кнопки, отображающей видео:

```
<Button MaxHeight="150" MaxWidth="200">
<Button.Content>
<MediaElement Source="Wildlife.wmv"></MediaElement>
</Button.Content>
</Button>
```

Текстовые элементы управления

В Windows Runtime выделяют три типа базовых элементов управления, позволяющих редактировать текст:

– TextBox – позволяет вводить простой текст в виде одной или нескольких строк;

– PasswordBox – текст скрывается с помощью специальных символов;

– RichEditBox – позволяет вводить форматированный текст. Если рассматривать свойства таких элементов, как TextBox и PasswordBox, можно выделить:

– AcceptsReturn – позволяет выполнять перевод каретки при вводе текста;

– IsReadOnly – определяет, будет ли текстовое поле доступно для ввода;

– SelectedText – возвращает выделенный текст;

– SelectionLength – позволяет получить или задать размер текущего выделения;

– SelectionStart – позволяет получить или задать позицию символа, с которого нужно произвести выделение;

– SelectionBackground – определяет цвет фона выделенного текста;

– SelectionForeground – задаёт цвет шрифта выделенного текста;

– TextWraping – определяет, будет ли текст переходить на другую строку, если он не помещается в видимой части одной строки.

<TextBox AcceptsReturn="True" TextWrapping="Wrap" Width="300" Height="200">

</TextBox>

Элементу TextBox можно противопоставить элемент управления TextBlock, задачей которого является отображение текста. Оба эти элемента способны удовлетворить все нужды разработчика в редактировании и отображении простого текста.

Если же говорить о форматированном тексте, тут требуется применять RichTextEdit и RichTextBlock. Если первый элемент позволяет ввести и редактировать форматированный текст, то второй – отображает только чтение. Оба элемента являются контейнерами для других элементов. Так, основным наполнением этих элементов выступает набор элементов Paragraph, которые описывают параграфы текстового документа. В свою очередь, параграф может включать набор из следующих элементов:

– Run – задаёт обычный текст;

– Span – служит для группировки других элементов;

– Bold – определяет жирное начертание символов;

– LineBreak – задаёт переход на другую строку;

– Italic – определяет рукописное начертание символов;

– Underline – выделяет текст подчёркиванием;

– HyperLink – создаёт гиперссылку, которая становится активной только в режиме ReadOnly элемента RichTextArea;

– InlineUIContainer – позволяет вставить в документ любой из элементов, порождённых UIElement.

Ниже приведён пример элемента RichTextBlock, отображающего текст с минимальным форматированием:

```
<RichTextBlock HorizontalAlignment="Left"
Name="rArea" VerticalAlignment="Top"
Height="300" Width="400" >
<Paragraph>
<Bold>This is a bold text</Bold>
<LineBreak></LineBreak>
<Underline>This is an underline text</Underline>
<LineBreak></LineBreak>
<Italic>This is an italic text</Italic>
<LineBreak></LineBreak>
This is a button:
<InlineUIContainer>
<Button Content="Button"></Button>
</InlineUIContainer>
</Paragraph>
</RichTextBlock>
```

Рис. 7.3. Элемент RichTextBlock

Результат работы приведён на рис. 7.3.

Естественно, создавать и заполнять RichTextBlock можно и с помощью кода на C++. Вот небольшой пример кода:

```
Bold b = new Bold();
b.Inlines.Add("This is a bold text");
Italic i = new Italic();
i.Inlines.Add("This is an italic text");
Underline u = new Underline();
u.Inlines.Add("This is an underlined text");
Paragraph myPar = new Paragraph();
myPar.Inlines.Add(b);
myPar.Inlines.Add(new LineBreak());
myPar.Inlines.Add(i);
myPar.Inlines.Add(new LineBreak());
myPar.Inlines.Add(u);
rArea.Blocks.Add(myPar);
```

Понятие стиля

Стиль – это специальный механизм, который позволяет собрать в одном месте все общие свойства определённой группы элементов управления. Очевидно, что если создаётся интерфейс, то все кнопки, поля редактирования, надписи и т.д. выполнены в едином стиле. Вряд ли кнопки в приложении раскрашены в разные цвета, а надписи имеют разный шрифт.

Рассмотрим небольшой пример, демонстрирующий создание четырёх кнопок.

```
<StackPanel>
<Button Width="100" Height="50"
Background="Green"
```

```
Content="Button 1" Margin="5" FontFamily="Arial"
FontSize="12" FontWeight="Bold"
Foreground="Blue" BorderThickness="3">
</Button>
<Button Width="100" Height="50"
Background="Green"
Content="Button 2" Margin="5" FontFamily="Arial"
FontSize="12" FontWeight="Bold"
Foreground="Blue" BorderThickness="3">
</Button>
<Button Width="100" Height="50"
Background="Green"
Content="Button 3" Margin="5" FontFamily="Arial"
FontSize="12" FontWeight="Bold"
Foreground="Blue" BorderThickness="3">
</Button>
<Button Width="100" Height="50"
Background="Green"
Content="Button 4" Margin="5" FontFamily="Arial"
FontSize="12" FontWeight="Bold"
Foreground="Blue" BorderThickness="3">
</Button>
</StackPanel>
```

Таким образом, создав всего четыре кнопки, продублировали множество строк кода. Это не только значительно увеличивает размер файла, но и ведёт к возникновению ошибок. Ведь указав неверно один атрибут, можно испортить внешний вид всего интерфейса.

Для выделения общих свойств в отдельный стиль в XAML используется специальный элемент Style. Поскольку он не должен входить в дерево элементов, а лишь задаёт набор свойств группы элементов, то располагать его следует внутри ресурсов приложения, страницы или отдельного элемента (в зависимости от области применения).

Продемонстрируем работу элемента Style на примере. Для этого определим следующий блок кода сразу после открывающего элемента:

```
Page.
<Page.Resources>
<Style x:Key="buttonStyle" TargetType="Button">
<Setter Property="Background" Value="Green"></Setter>
<Setter Property="Margin" Value="5"></Setter>
<Setter Property="FontFamily" Value="Arial"></Setter>
<Setter Property="FontSize" Value="12"></Setter>
<Setter Property="FontWeight" Value="Bold"></Setter>
<Setter Property="Foreground" Value="Blue"></Setter>
<Setter Property="BorderThickness" Value="3"></Setter>
</Style>
</Page.Resources>
```

Чтобы задать стиль, необходимо указать ключ и тип элементов, для которых будет применяться стиль. Ключ используется для поиска необходимого стиля, а TargetType определяет, к каким элементам применим стиль. Далее с помощью набора элементов Setter мы перечислили все свойства и их значения. Код XAML интерфейса выглядит следующим образом:

```
<StackPanel>
<Button Width="100" Height="50"
Style="{StaticResource buttonStyle}" Content="Button 1">
</Button>
<Button Width="100" Height="50"
Style="{StaticResource buttonStyle}" Content="Button 2">
</Button>
<Button Width="100" Height="50"
Style="{StaticResource buttonStyle}" Content="Button 3">
</Button>
<Button Width="100" Height="50"
Style="{StaticResource buttonStyle}" Content="Button 4">
</Button>
</StackPanel>
```

Стиль может быть и неявным, т.е. не содержать ключа. В таком случае свойства, определённые в стиле, будут применяться для всех элементов указанного типа (если элемент не задаёт указанные свойства явно). Вот как будет выглядеть код, определяющий неявный стиль для наших кнопок:

```
<Page.Resources>
<Style TargetType="Button">
<Setter Property="Background" Value="Green"></Setter>
<Setter Property="Margin" Value="5"></Setter>
<Setter Property="FontFamily" Value="Arial"></Setter>
<Setter Property="FontSize" Value="12"></Setter>
<Setter Property="FontWeight" Value="Bold"></Setter>
<Setter Property="Foreground" Value="Blue"></Setter>
<Setter Property="BorderThickness" Value="3"></Setter>
</Style>
</Page.Resources>
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
<StackPanel>
<Button Width="100" Height="50"
Content="Button 1">
</Button>
<Button Width="100" Height="50"
Content="Button 2">
</Button>
<Button Width="100" Height="50"
Content="Button 3">
</Button>
<Button Width="100" Height="50"
```

```
Content="Button 4">
</Button>
</StackPanel>
</Grid>
```

Стили представляют собой очень мощный механизм, позволяющий сделать не только Ваше приложение в едином формате, но и придать общность большинству запускаемых приложений. Чтобы у разработчика была возможность создавать страницы, придерживаясь единого стиля, Visual Studio включает в любой проект файл StandardStyles.xaml, который определяет набор стандартных стилей на все случаи жизни. Так, попробуйте модифицировать код выше, прописав кнопкам стиль BackButtonStyle.

```
<StackPanel>
<Button Width="100" Height="50"
Content="Button 1" Style="{StaticResource BackButtonStyle}">
</Button>
<Button Width="100" Height="50"
Content="Button 2" Style="{StaticResource BackButtonStyle}">
</Button>
<Button Width="100" Height="50"
Content="Button 3" Style="{StaticResource BackButtonStyle}">
</Button>
<Button Width="100" Height="50"
Content="Button 4" Style="{StaticResource BackButtonStyle}">
</Button>
</StackPanel>
```

BasedOn-стили

XAML в Windows Runtime поддерживает и BasedOn-стили. Это позволяет задать общие стили применительно к базовым классам для общей группы элементов управления (или для конкретного типа элементов управления) и определить дополнительные стили путём наследования. Ниже приведён пример BasedOn-стиля.

```
<Style x:Key="baseStyle" TargetType="Button" >
<Setter Property="Width" Value="100"></Setter>
</Style>
<Style x:Key="btnStyleRus"
BasedOn="{StaticResource baseStyle}" TargetType="Button">
<Setter Property="Foreground" Value="Blue"></Setter>
<Setter Property="Content" Value="Кнопка"></Setter>
</Style>
<Style x:Key="btnStyle"
BasedOn="{StaticResource baseStyle}" TargetType="Button">
<Setter Property="Foreground" Value="Green"></Setter>
<Setter Property="Content" Value="Button"></Setter>
</Style>
```

Пример не совсем удачный (локализировать приложения таким образом нельзя), однако общую идею он демонстрирует.

Ресурсы

Выше говорилось о возможности создания стилей. Которые, как было отмечено, не должны входить в дерево XAML-элементов. Все стили записывали внутри свойства Resources, принадлежащего объекту Page. Подобное свойство есть у всех объектов XAML, порождённых от FrameworkElement, а также у объекта приложения (Application). Благодаря ресурсам разработчик может сохранять стили и другие части XAML-кода до востребования. Так, в ресурсы можно выделять объекты некоторых типов, стили, шаблоны данных и др. Фактически свойство Resources ссылается на некий словарь, который может быть доступен как из кода, так и из XAML.

Если используются ресурсы на уровне приложения, то доступ к ресурсу может быть осуществлён лишь по ключу, вне зависимости от того, в каком из родительских элементов объявлен ресурс. Если же ресурс был объявлен в одном из дочерних элементов, то он не может использоваться родительскими элементами.

Рассмотрим небольшой пример.

```
<Page.Resources>
<LinearGradientBrush x:Key="myBrush">
<GradientStop Color="Red" Offset="0"></GradientStop>
<GradientStop Color="Green" Offset="1"></GradientStop>
</LinearGradientBrush>
</Page.Resources>
<StackPanel>
<Button Width="100" Height="50"
Background="{StaticResource myBrush}"
Content="Button 1" Margin="5">
</Button>
</StackPanel>
```

Здесь определили в роли ресурса градиентную кисть, указав в качестве ключа (он же является именем) myBrush. Данный ключ будет использоваться для доступа к ресурсу, который определяется с помощью расширения разметки StaticResource. Достаточно указать имя ключа в качестве параметра.

В примере выше разместили ресурсы внутрь элемента Page, который является контейнером для интерфейса. Это обеспечивает видимость ресурсов для любого дочернего элемента. Если мы хотим ограничить область видимости, то ресурсы можно разместить в любом из дочерних элементов, например в StackPanel.

```
<StackPanel>
<StackPanel.Resources>
<LinearGradientBrush x:Key="myBrush">
```

```
<GradientStop Color="Red" Offset="0"></GradientStop>
<GradientStop Color="Green" Offset="1"></GradientStop>
</LinearGradientBrush>
</StackPanel.Resources>
<Button Width="100" Height="50"
Background="{StaticResource myBrush}"
Content="Button 1" Margin="5">
</Button>
</StackPanel>
```

Таким образом, для конечного элемента управления нет никакой разницы, где находятся ресурсы. Главное, чтобы они были в области видимости элемента, а доступ к ним осуществляется по одинаковой схеме.

Наконец, если необходимо обеспечить ресурсам глобальную область видимости, то можете разместить их на уровне всего приложения. Такой подход очень эффективен при разработке сложных приложений, т.е. приложений, обладающих несколькими окнами и механизмом навигации между ними. Вот пример определения этих же ресурсов внутри файла приложения.

```
<Application
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:Application4">
<Application.Resources>
<LinearGradientBrush x:Key="myBrush">
<GradientStop Color="Red" Offset="0"></GradientStop>
<GradientStop Color="Green" Offset="1"></GradientStop>
</LinearGradientBrush>
</Application.Resources>
</Application>
```

Выделение ресурсов объектов в отдельные файлы

Ресурсы можно хранить в отдельных файлах и собирать в нужном месте с помощью элемента ResourceDictionary. Вынесем кисть из предыдущего примера в отдельный файл RD1.xaml.

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<LinearGradientBrush x:Key="myBrush">
<GradientStop Color="Red" Offset="0"></GradientStop>
<GradientStop Color="Green" Offset="1"></GradientStop>
</LinearGradientBrush>
</ResourceDictionary>
```

Для подключения внешнего файла к интерфейсу приложения используется всё тот же ResourceDictionary.

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

```
xmlns:local="using:Application4">
<Application.Resources>
<ResourceDictionary>
<ResourceDictionary.MergedDictionaries>
<ResourceDictionary Source="RD1.xaml" />
</ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
</Application.Resources>
</Application>
```

Если у вас несколько файлов – не проблема. Перечислите их все. Обратите внимание, что стандартный файл StandardStyles.xaml как раз и оформлен как внешний файл ресурсов, который подключается внутри файла App.xaml.

Если Вы хотите подключить файл с ресурсами в ресурсы одного из элементов, то это делается аналогичным образом.

8. ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННОЙ СИСТЕМЫ ДЛЯ WINDOWS STORE. ОБРАБОТКА ДАННЫХ

Привязка к свойству элемента управления

Начнём с того, что для реализации привязки используется объект типа Binding. Независимо от того, связываются элементы или элемент и данные, всегда используется именно Binding. При этом Binding можно совершенно спокойно использовать как в коде, так и в разметке XAML.

Естественно, что использование Binding в XAML – самая распространённая ситуация. Для этих целей в XAML существует специальное расширение разметки. Рассмотрим небольшой пример:

```
<Page x:Class="Chapter5_Binding.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}"
VerticalAlignment="Center">
<StackPanel x:Name="LayoutRoot">
<Image Source="Assets/hydrangeas.jpg" Width="400">
<Image.Projection>
<PlaneProjection RotationY=
"{Binding Value, ElementName=slider}">
</PlaneProjection>
</Image.Projection>
</Image>
<Slider Minimum="0" Maximum="360" Name="slider"
Width="400" Margin="10"></Slider>
</StackPanel>
</Grid>
</Page>
```

Здесь создали элемент управления Image, который хотим «вращать» по оси Y. Для создания эффекта размещения элемента в трёхмерном пространстве используется объект PlaneProjection, содержащий свойство RotationY, которое задаёт угол поворота элемента по оси Y. Чтобы сделать интерфейс более динамичным, вторым элементом добавим ползунок, который и будет задавать угол поворота. Используем два параметра:

– Path – позволяет задать свойство источника, с которым происходит связывание. Поскольку это свойство является свойством по умолчанию, то явно Path можно не писать;

– ElementName – задаёт имя элемента-источника.

В данном примере независимо от того, как модифицируется свойство Value бегунка, свойство RotationY будет обновляться автоматически. Привязку можно реализовать и по-другому:

```
<Page x:Class="Chapter7_Binding.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}"
VerticalAlignment="Center">
<StackPanel x:Name="LayoutRoot" >
<Image Source="Hydrangeas.jpg" Width="400">
<Image.Projection>
<PlaneProjection x:Name="projection"></PlaneProjection>
</Image.Projection>
</Image>
<Slider Minimum="0" Maximum="360" Name="slider"
Width="400" Margin="10"
Value=
"{Binding RotationY, ElementName=projection, Mode=TwoWay}">
</Slider>
</StackPanel>
</Grid>
</Page>
```

В этом примере в качестве источника выступает изображение. Отличие состоит в том, что, выполняя привязку ползунка к изображению, мы указали дополнительное свойство Mode. Это свойство может принимать одно из трёх значений:

– OneTime – значение свойства устанавливается на основании значения свойства источника, но установка происходит лишь в момент создания объектов. Любые изменения в будущем игнорируются;

– OneWay – значение свойства устанавливается на основании значения свойства источника. При изменении свойства источника будет обновляться свойство основного объекта;

– TwoWay – значение свойства устанавливается на основании значения свойства источника. При изменении свойства источника или свойства основного объекта будут происходить взаимные обновления.

В примере установили значение TwoWay свойству Mode. Таким образом, несмотря на то, что картинка является источником, её поворот успешно задаётся ползунком.

Выбор источника зависит от конкретного приложения. Так, в примере выше выбор источника был не принципиален, но если мы решим добавить дополнительный элемент, модифицирующий значение угла поворота, то установить свойству два элемента Binding нам не удастся, а вот изменить направление привязки можно. Пример ниже расширяет наш интерфейс текстовым полем, которое также задаёт угол поворота.

```
<Page x:Class="Chapter5_Binding.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}"
VerticalAlignment="Center">
<StackPanel x:Name="LayoutRoot">
<Image Source="Assets/hydrangeas.jpg" Width="400">
<Image.Projection>
<PlaneProjection x:Name="projection"></PlaneProjection>
</Image.Projection>
</Image>
<Slider Minimum="0" Maximum="360" Name="slider"
Width="400" Margin="10"
Value=
"{Binding RotationY, ElementName=projection, Mode=TwoWay}">
</Slider>
<TextBox Width="200" Text=
"{Binding RotationY, ElementName=projection,
Mode=TwoWay}"></TextBox>
</StackPanel>
</Grid>
</Page>
```

Обратите внимание, что в этом примере действие значения, введённого в TextBox, вступит в силу лишь при потере фокуса элементом TextBox. Это связано с тем, что поведение свойств при реализации привязки может отличаться. Чтобы задать поведение свойства, используют метаданные (атрибуты). В свою очередь, метаданные для TextBox заданы таким образом, что им нужна потеря фокуса текстового поля, чтобы произвести обновление. Для элемента TextBox это вполне обосновано.

Наконец, если необходимо установить привязку к данным в коде, то это также можно сделать без проблем. Вот как будет выглядеть код для TextBox из нашего примера:

```
Binding binding = new Binding();
binding.ElementName = "projection";
binding.Path = new PropertyPath("RotationY");
binding.Mode = BindingMode.TwoWay;
txtBox.SetBinding(TextBox.TextProperty, binding);
```

Перейдём теперь от привязки к элементам к привязке к объектам, которые не являются элементами управления.

Привязка к объекту

На практике значительно чаще приходится выполнять привязку элементов к данным, которые содержатся в объекте некоторого класса. Эти данные могут быть получены из базы данных или сгенерированы во время работы приложения. Создадим простой класс, описывающий информацию о сотруднике:

```
public class Employee
{
public string FirstName
{
get;   set;
}
public string LastName
{
get;  set;
}
public string EMail
{
get;   set;
}
public int Age
{
get;   set;
} }
```

Этот класс содержит четыре свойства с модификатором public. Это основное требование при привязке к данным: свойства объектов, которые выступают в качестве источника, должны быть общедоступными.

Следующий код создаёт простую форму и связывает эту форму с объектом типа Employee, который мы определили в ресурсах (аналогично можно определить и в коде):

```
<Page x:Class=" Chapter5_Binding.MainPage "
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="using:Element_Binding_Sample">
<Page.Resources>
<local:Employee x:Key="employee" Age="27" FirstName="Sergii"
LastName="Lutai" EMail="sergii.lutai@dct.ua" />
</Page.Resources>
<Grid x:Name="LayoutRoot" Width="400">
<Grid.RowDefi nitions>
<RowDefi nition></RowDefi nition>
<RowDefi nition></RowDefi nition>
<RowDefi nition></RowDefi nition>
```

```
<RowDefi nition></RowDefi nition>
</Grid.RowDefi nitions>
<Grid.ColumnDefi nitions>
<ColumnDefi nition></ColumnDefi nition>
<ColumnDefi nition></ColumnDefi nition>
</Grid.ColumnDefi nitions>
<TextBlock Text="First Name:" Grid.Row="0" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding FirstName, Source={StaticResource
employee}, Mode=TwoWay}"
Grid.Row="0" Grid.Column="1">
</TextBox>
<TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding LastName, Source={StaticResource
employee}, Mode=TwoWay}"
Grid.Row="1" Grid.Column="1">
</TextBox>
<TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding EMail, Source={StaticResource
employee}, Mode=TwoWay}"
Grid.Row="2" Grid.Column="1">
</TextBox>
<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding Age, Source={StaticResource
employee}, Mode=TwoWay}"
Grid.Row="3" Grid.Column="1">
</TextBox>
</Grid>
</Page>
```

Используемый механизм точно такой, как и при привязке к элементам. Только вместо ElementName используется свойство Source, которое задаёт ссылку на объект (в данном случае выбираемый из ресурсов).

Модифицируем код выше, используя ещё одно полезное свойство – DataContext:

```
<Page
x:Class="Chapter5_Binding.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="using:Element_Binding_Sample">
<Page.Resources>
<local:Employee x:Key="employee" Age="27" FirstName="Sergii"
LastName="Lutai" EMail="sergii.lutai@dct.ua" />
</Page.Resources>
```

```
<Grid x:Name="LayoutRoot" Width="400" DataContext="{StaticResource
employee}">
<Grid.RowDefi nitions>
<RowDefi nition></RowDefi nition>
<RowDefi nition></RowDefi nition>
<RowDefi nition></RowDefi nition>
<RowDefi nition></RowDefi nition>
</Grid.RowDefi nitions>
<Grid.ColumnDefi nitions>
<ColumnDefi nition></ColumnDefi nition>
<ColumnDefi nition></ColumnDefi nition>
</Grid.ColumnDefi nitions>
<TextBlock Text="First Name:" Grid.Row="0" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding FirstName, Mode=TwoWay}"
Grid.Row="0" Grid.Column="1">
</TextBox>
<TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding LastName, Mode=TwoWay}" Grid.Row="1"
Grid.Column="1">
</TextBox>
<TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding EMail, Mode=TwoWay}" Grid.Row="2"
Grid.Column="1">
</TextBox>
<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding Age, Mode=TwoWay}"
Grid.Row="3" Grid.Column="1">
</TextBox>
</Grid>
</Page>
```

Код в примере проще. Дело в том, что если не указать имя объекта с данными явно, то механизм привязки начинает перебор свойств DataContext всех родительских элементов, пока не найдёт непустой. Найденный DataContext – механизм связывания, используется в качестве источника.

Привязка к коллекции

При привязке простых свойств обычно не возникает проблем. Но привязка коллекций требует дополнительных усилий. Расширим класс Employee, перегрузив метод ToString:

```
public override string ToString()
{
return String.Format("{0} {1}", fi rstName, lastName);
}
```

Этот метод нам понадобится при заполнении одного из элементов управления коллекцией элементов типа Employee.

Коллекции способны принимать только те элементы, которые являются наследниками ItemsControl. Среди таких элементов можно выделить и ListBox, который способен отображать список элементов.

Прежде чем писать код, необходимо определиться со свойствами, которые используются при привязке коллекции к элементу. Тут существуют три свойства:

– ItemsSource – задаёт коллекцию элементов. Если не задан DisplayMemberPath, то каждый элемент попытается вызвать метод ToString. Вот почему мы и предопределили ToString;

– ItemTemplate – используется для задания шаблона отображения конкретного элемента;

– DisplayMemberPath – задаёт конкретное свойство для отображения в элементе управления.

Теперь перейдём к коду. Сначала расширим интерфейс, отобразив ListBox:

```
<Page x:Class="Chapter5_Binding.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:data="using:Chapter5_Binding">
<StackPanel>
<ListBox Height="300" Name="lstEmp"></ListBox>
<Grid x:Name="LayoutRoot" Background="White" Width="400"
DataContext="{Binding SelectedItem, ElementName=lstEmp}">
<Grid.RowDefi nitions>
<RowDefi nition Height="Auto"></RowDefi nition>
<RowDefi nition Height="Auto"></RowDefi nition>
<RowDefi nition Height="Auto"></RowDefi nition>
<RowDefi nition Height="Auto"></RowDefi nition>
</Grid.RowDefi nitions>
<Grid.ColumnDefi nitions>
<ColumnDefi nition></ColumnDefi nition>
<ColumnDefi nition></ColumnDefi nition>
</Grid.ColumnDefi nitions>
<TextBlock Text="First Name:" Grid.Row="0" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding Path=FirstName}" Grid.Row="0"
Grid.Column="1">
</TextBox>
<TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding Path=LastName}" Grid.Row="1"
Grid.Column="1">
</TextBox>
```

```
<TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding Path=EMail}" Grid.Row="2"
Grid.Column="1">
</TextBox>
<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0">
</TextBlock>
<TextBox Text="{Binding Path=Age}" Grid.Row="3"
Grid.Column="1">
</TextBox>
</Grid>
</StackPanel>
</Page>
```

Контекст для формы с детальными данными выбираем из элемента ListBox, используя привязку (рис. 8.1). Теперь реализуем код, создающий список элементов. В качестве структуры данных можно выбрать любой класс – главное, чтобы он реализовывал интерфейс IEnumerable.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
List<Employee> employeeList = new List<Employee>();
Employee emp = new Employee();
emp.FirstName = "Sergiy";
emp.LastName = "Baydachnyy";
emp.EMail = "sbaidachni@gmail.com";
emp.Age = 31;
employeeList.Add(emp);
emp = new Employee();
emp.FirstName = "Sergii";
emp.LastName = "Lutai";
emp.EMail = "sergii.lutai@dct.ua";
emp.Age = 27;
employeeList.Add(emp);
lstEmp.ItemsSource = employeeList;
}
```

Рис. 8.1. Элемент ListBox

Следует отметить, что вместо класса List лучше использовать коллекцию ObservableCollection (рис. 8.2). Дело в том, что если Вы решите добавлять или удалять элементы из списка, то столкнётесь с необходимостью реализации интерфейса INotifyCollectionChanged. A ObservableCollection уже реализует этот интерфейс.

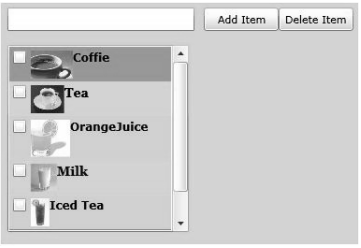

Рис. 8.2. Коллекция ObservableCollection

Вот ещё пример.

```
<ListBoxItem
Background="LightCoral"
Foreground="Red"
```

```
            FontFamily="Verdana" FontSize="12" FontWeight="Bold">
        <CheckBox Name="CoffieCheckBox">
            <StackPanel Orientation="Horizontal">
            <Image Source="coffie.jpg" Height="30"></Image>
            <TextBlock Text="Coffie"></TextBlock>
        </StackPanel>
    </CheckBox>
</ListBoxItem>
<ListBoxItem Background="LightGray" Foreground="Black"
            FontFamily="Georgia" FontSize="14" FontWeight="Bold">
    <CheckBox Name="TeaCheckBox">
        <StackPanel Orientation="Horizontal">
            <Image Source="tea.jpg" Height="30"></Image>
            <TextBlock Text="Tea"></TextBlock>
        </StackPanel>
    </CheckBox>
</ListBoxItem>
<ListBoxItem Background="LightBlue" Foreground="Purple"
            FontFamily="Verdana" FontSize="12" FontWeight="Bold">
    <CheckBox Name="OrangeJuiceCheckBox">
        <StackPanel Orientation="Horizontal">
            <Image Source="OrangeJuice.jpg" Height="40"></Image>
            <TextBlock Text="OrangeJuice"></TextBlock>
        </StackPanel>
    </CheckBox>
</ListBoxItem>
<ListBoxItem Background="LightGreen" Foreground="Green"
            FontFamily="Georgia" FontSize="14" FontWeight="Bold">
    <CheckBox Name="MilkCheckBox">
        <StackPanel Orientation="Horizontal">
            <Image Source="Milk.jpg" Height="30"></Image>
            <TextBlock Text="Milk"></TextBlock>
        </StackPanel>
    </CheckBox>
</ListBoxItem>
<ListBoxItem Background="LightBlue" Foreground="Blue"
            FontFamily="Verdana" FontSize="12" FontWeight="Bold">
    <CheckBox Name="IcedTeaCheckBox">
        <StackPanel Orientation="Horizontal">
            <Image Source="IcedTea.jpg" Height="30"></Image>
            <TextBlock Text="Iced Tea"></TextBlock>
        </StackPanel>
    </CheckBox>
</ListBoxItem>
<ListBoxItem Background="LightSlateGray" Foreground="Orange"
            FontFamily="Georgia" FontSize="14" FontWeight="Bold">
    <CheckBox Name="MangoShakeCheckBox">
        <StackPanel Orientation="Horizontal">
            <Image Source="MangoShake.jpg" Height="30"></Image>
```

```
        <TextBlock Text="Mango Shake"></TextBlock>
      </StackPanel>
    </CheckBox>
  </ListBoxItem>
```

Использование паттерна MVVM

Модель связывания данных в WinRT позволяет использовать паттерн MVVM для архитектуры приложения. Данный паттерн был представлен Джоном Госсманом в 2005 году. Он представляет модификацию паттерна MVP и ориентирован на современные платформы разработки, которые имеют возможность связывания данных.

Благодаря применению этого паттерна у разработчика появляется возможность разделения представления и модели, тем самым изменяя их независимо друг от друга. Например, разработчики создают логику работы приложения, а дизайнеры работают над интерфейсом приложения.

Данный паттерн следует применять, если в технологии присутствует механизм связывания данных. Использование паттерна MVC/MVP в таких случаях не вкладывается в их концепцию.

Паттерн MVVM делится на три части (рис. 8.3):

– Model – представляет фундаментальные данные, необходимые для работы приложения;

– View – это графический интерфейс нашего приложения;

– ViewModel – является, с одной стороны, абстракцией View, а с другой – предоставляет обёртку данных из Model, которые подлежат связыванию. То есть она содержит Model, которая преобразована к View, а также содержит команды, которыми может пользоваться View, чтобы влиять на Model.

Классическая реализация базовой ViewModel:

```
public class ViewModel : INotifyPropertyChanged
{
public event PropertyChangedEventHandler PropertyChanged;
protected void RaisePropertyChanged(string propertyName)
{
var handler = PropertyChanged;
if (handler != null)
{
handler(this, new PropertyChangedEventArgs(propertyName));
}}}
```

Далее, разделение логики на модели ViewModel происходит отдельно для каждого проекта. Все последующие ViewModel являются наследниками базовой.

Например, возьмём тип данных Employee и реализуем для него ViewModel public:

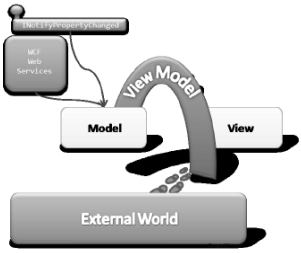

Рис. 8.3. Модель ViewModel

```
class EmployeeViewModel : ViewModel
{
public ObservableCollection<Employee>
EmployeeCollection { get; private set; }
public ICommand AddEmployee { get; private set; }
public ICommand EditEmployee { get; private set; }
public ICommand DeleteEmplyee { get; private set; }
public EmployeeViewModel()
{
EmployeeCollection = new ObservableCollection<Employee>
{
new Employee { FirstName="Sergii", LastName="Lutai",
Age=27, EMail="sergii.lutai@dct.ua"},
new Employee { FirstName="Sergiy", LastName="Baydachnyy",
Age=32, EMail="Sergiy.Baydachnyy@microsoft.com"},
new Employee { FirstName="Ivan", LastName="Ivanov",
Age=29, EMail="ivan@ivanov.ua"}
};
AddEmployee = new RelayCommand<Employee>
(this.AddEmployeeExecute, this.AddEmployeeCanExecute);
}
private void AddEmployeeExecute(Employee employee)
{
//TODO: Логика добавления объекта
}
private bool AddEmployeeCanExecute(Employee employee)
{
//TODO: Логика проверки возможности добавления нового объекта
return employee != null;
}
}
```

В данном примере класс RelayCommand является одной из возможных реализаций интерфейса System.Windows.Input.RelayCommand:

```
public class RelayCommand<T>: ICommand
{
private Action<T> action;
private Func<T, bool> func;
public RelayCommand(Action<T> action)
{
this.action = action;
}
public RelayCommand(Action<T> action, Func<T, bool> func)
{
this.action = action;
this.func = func;
}
public bool CanExecute(object parameter)
{
bool result = true;
```

```
T value = default(T);
if (parameter != null && !(parameter is T))
{
throw new ArgumentException(«Wrong parameter type.",
«parameter");
}
if (parameter != null)
{
value = (T)parameter;
}
if (func != null)
{
result = func(value);
}
return result;
}
public event EventHandler CanExecuteChanged;
public void Execute(object parameter)
{
T value = default(T);
if (parameter != null && !(parameter is T))
{
throw new ArgumentException(«Wrong parameter type.",
«parameter");
}
if (parameter != null)
{
value = (T)parameter;
}
var canExecute = CanExecute(value);
if (canExecute)
{
action(value);
}
}
public void RaiseCanExecuteChanged()
{
var handler = CanExecuteChanged;
if (handler != null)
{
handler(this, EventArgs.Empty);
}
}
}
```

Для проверки нашей логики будем использовать следующее представление:

```
<Page
x:Class="MVVM.Sample.BlankPage"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:MVVM.Sample"
xmlns:viewModel="using:ViewModel">
<Page.Resources>
<viewModel:EmployeeViewModel x:Key="EmployeeViewModel"/>
</Page.Resources>
<Page.DataContext>
<Binding Source="{StaticResource EmployeeViewModel}"/>
</Page.DataContext>
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
<Button Content="Add New Employee" VerticalAlignment="Top"
Command="{Binding AddEmployee, Mode=OneTime,
Source={StaticResource EmployeeViewModel}}"/>
<ListBox ItemsSource="{Binding EmployeeCollection}"
Margin="0,40,0,0">
<ListBox.ItemTemplate>
<DataTemplate>
<StackPanel Orientation="Horizontal">
<Button Content="Delete"
Command="{Binding DeleteEmplyee, Mode=OneWay,
Source={StaticResource EmployeeViewModel}}"
CommandParameter="{Binding Mode=OneWay}"/>
<Button Content="Edit"
Command="{Binding EditEmployee, Mode=OneWay,
Source={StaticResource EmployeeViewModel}}"
CommandParameter="{Binding Mode=OneWay}"/>
<TextBlock Text="{Binding FirstName}"
Margin="40,0,40,0"/>
<TextBlock Text="{Binding LastName}"/>
</StackPanel>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
</Grid>
</Page>
```

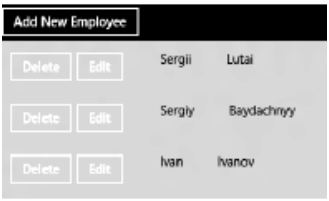

Рис. 8.4. Пример реализации View Model

Рассмотренный выше пример показан на рис. 8.4.

Таким образом, используя паттерн MVVM, можно разделить работу дизайнеров и разработчиков, что в свою очередь позволяет сосредоточиться именно на логике бизнес-процессов внутри приложения. При этом возможность тестирования бизнес-логики приложения увеличивается в разы.

9. РАЗРАБОТКА И РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ ДЛЯ WINDOWS STORE

Все объекты в приложении Windows Store связаны между собой и передаеют друг другу управление. Такие отношения называются навигацией. Приложения для Windows Store поддерживают несколько типов навигации:

– иерархическая система (рис. 9.1);
– плоская система.

В большинстве приложений для Windows Store в Windows 8 используется иерархическая система навигации. Такая структура используется повсеместно и хорошо знакома по предшествующим версиям Windows, но она стала удобнее благодаря внедрению узловой структуры.

Данная система навигации делает приложения для Windows Store быстрыми и динамичными, сохраняя при этом простоту использования. Лучше всего подходит для приложений с крупными коллекциями содержимого или большим количеством разделов содержимого, предназначенного для исследования пользователем.

Суть узловой навигации – в распределении содержимого по различным разделам и по степени развёрнутости.

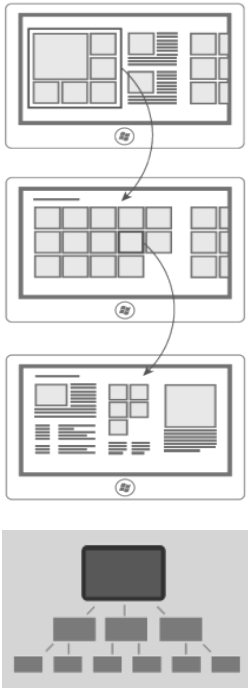

Рис. 9.1. Иерархическая система навигации

Страницы узлов

Страницы узлов – это точки входа пользователя в приложение. Здесь содержимое отображается с развёртыванием по горизонтали и акцентирует внимание пользователей на новых и доступных элементах.

Узел состоит из различных категорий содержимого, каждый из которых связан со страницами разделов приложения. Каждый раздел должен отображать содержимое или функцию. Узел должен содержать в себе множество визуальных элементов, быть привлекательным для пользователей и направлять их в различные части приложения.

Страницы разделов

Страницы разделов – это второй уровень приложения. Здесь содержимое может быть отображено в любом виде, который наилучшим образом отражает сценарий и содержимое раздела.

Страница раздела состоит из отдельных элементов, у каждого из которых есть своя страница сведений. На страницах разделов также используются группирование и панорамная структура.

Рис. 9.2

Страницы сведений

Страницы сведений – это третий уровень приложения. Здесь отображаются сведения об отдельных элементах, причём формат этих сведений во многом зависит от типа содержимого.

Страница сведений содержит сведения об элементе или функцию. Страницы сведений могут содержать множество сведений или отдельный объект, например изображение или видеоролик (рис. 9.2).

Во многих приложениях для Windows Store используется плоская система навигации. Такая структура распространена в играх, браузерах и офисных приложениях, где пользователь переходит между страницами, вкладками или режимами, которые находятся на одном и том же иерархическом уровне.

Эта структура отлично подходит в случае, когда ключевой сценарий подразумевает быстрое переключение между небольшим количеством страниц и вкладок.

Суть плоской системы заключается в распределении содержимого по различным страницам.

Реализация навигации

Рассмотрим приложение в Windows 8 способно отобразить контент. Речь идёт именно о контенте, так как разработка приложений для Windows Store базируется на отсутствии оболочки у окна. Тем не менее окно в приложении присутствует, оно создаётся неявно, но на него можно получить ссылку, используя свойство Current класса Window. Получив ссылку на окно приложения, разработчик может установить свойство Content, которое будет определять содержимое окна. Когда же содержимое полностью сформировано, его можно отобразить с помощью метода Activate. Иными словами, если в Вашем приложении предусмотрен переход на другие страницы, то этого можно достичь, установив свойство Content в ссылку на страницу и вызвав Activate. В сентябрьской версии Windows 8 (Platform Preview) нечто подобное было реализовано. Проблема лишь в том, что данный механизм не позволяет сохранять историю переходов, не поддерживает хорошего способа передачи параметров и т.д. Поэтому для организации навигации необходимо использовать специальный контейнер Frame, который предназначен для поддержки контекста навигации.

Заглянем внутрь файла App.xaml.cs.

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
 var rootFrame = new Frame();
 rootFrame.Navigate(typeof(BlankPage));
 Window.Current.Content = rootFrame;
 Window.Current.Activate();
}
```

При запуске приложения создаётся объект типа Frame, который устанавливается в свойство Content. С помощью метода Navigate объект типа Frame загружает страницу BlankPage (имя страницы по умолчанию), что позволяет отобразить её после вызова метода окна Activate. Таким образом, Frame является тем классом, который обеспечивает контекст навигации, и дает доступ к ряду методов и свойств, связанных с навигацией:

– CacheSize – указывает количество страниц, сохраняемых в кэше при «глубокой» навигации;

– CanGoBack – определяет, возможно ли осуществить навигацию на предыдущую страницу (была ли и есть ли страница в кэше);

– CanGoForward – определяет, можно ли совершить навигацию на страницу, которую пользователь уже посещал и с которой вернулся на предыдущую;

– GoBack – выполняет навигацию на предыдущую страницу;

– GoForward – выполняет навигацию на следующую страницу;

– Navigated – событие, генерируемое после успешной навигации;

– Navigating – событие, которое генерируется перед попыткой навигации;

– NavigationFailed – событие, которое возникает, если во время навигации произошла ошибка (например, страница отсутствует);

– NavigationStopped – событие, которое генерируется в тот момент, когда текущая навигация была прервана с помощью метода StopLoading или путём выполнения другой навигации;

– StopLoading – завершает загрузку данных и прекращает навигацию на указанную страницу;

– Navigate – самый важный метод, который позволяет выполнять навигацию на указанную страницу.

Перечисленные методы достаточно просты, поэтому реализация навигации не составляет труда. Если говорить об использовании этих методов в сгенерированном коде, рекомендуется обратиться к файлу LayoutAwarePage.cs, который генерируется при создании любого проекта.

```
protected virtual void GoHome(object sender, RoutedEventArgs e)
{
 if (this.Frame != null)
 {
 while (this.Frame.CanGoBack) this.Frame.GoBack();
 }
}
```

Тут описан обработчик события, которое возникает при нажатии кнопки Home на панели приложения. Для перехода на основную страницу совершена попытка вернуться на главную вследствие пролистывания всех страниц в обратном порядке. При этом объект типа Frame можно получить через одноимённое свойство объекта Page, а потом осуществлять навигацию.

При разработке приложений необходимо учитывать особенности среды, в которой будет происходить взаимодействие пользователя с приложением и по возможности максимально интегрировать свой продукт с платформой. Это поможет создать ощущение, что продукт является неотъемлемой частью платформы, для которой вы его разрабатываете.

Создавая приложения для Windows 8, возможно использовать различные уведомления (Tile, Toast), интеграцию с поиском, делиться информацией со своими друзьями через социальные сервисы и удобно настраивать приложения под свои нужды. В этой главе мы рассмотрим возможности интеграции наших приложений с Windows 8.

Работа с плиткой

Плитка является неотъемлемой частью приложения для Windows Store и позволяет пользователям запускать приложения со стартового экрана ОС. По умолчанию плитка – статическое изображение. Но мы можем добавить ему динамики, тем самым заинтересовав пользователя, сообщая об изменениях, когда приложение находится в неактивном состоянии. Динамические плитки позволяют Windows 8 казаться «живой».

Для приложений мы можем использовать два режима отображения плиток (рис. 9.3). Пользователь сам выбирает, какой из них использовать на стартовом экране:

– Стандартный – может содержать бренд-приложения – название или логотип продукта, а также значок. Из-за того, что данный режим отображения плитки может содержать ограниченный набор контента, доступен только один шаблон для его создания;

– Расширенный – этот режим отображения плитки может содержать те же данные, что и стандартный режим, а также ещё более детализированное описание и визуально интригующее оформление. В этом режиме существует широкий выбор шаблонов, которые позволят сделать контент более привлекательным. При использовании этого режима приложение обязательно должно иметь плитку стандартного размера, чтобы пользователь мог в любой момент переключиться на компактную плитку.

Содержимое плитки определяется в документе XML, который выбирается из множества шаблонов. Плитка может содержать изображение и текст, в зависимости от того, какой шаблон был выбран, а также отображать значок, логотип или короткое имя приложения (см. рис. 9.3). Значок отображается в правом нижнем углу плитки. Короткое имя располагается в левом нижнем углу. В манифесте приложения мы выбираем, что хотим отображать: короткое имя приложения или его логотип.

Рис. 9.3. Работа с расширенной плиткой

Рис. 9.4. Компактная плитка на рабочем столе

При настройке плитки можно указать возможность повторения до пяти последних уведомлений (рис. 9.4). Каждому уведомлению можно присваивать идентификатор, с помощью которого система будет решать, необходимы ли обновление или замена одного из существующих уведомлений на клиенте. Если Windows находит уведомление с таким же идентификатором, происходит замена. Если такого идентификатора в очереди нет, то из неё вытесняется наиболее старое уведомление.

Когда приложение устанавливается в первый раз, на стартовом экране отображается плитка приложения по умолчанию. Это статическая картинка определена в манифесте приложения; обычно она отображает логотип продукта.

Эта картинка заменяется только в случае отправки соответствующего уведомления. Фактически мы создаём плитки при указании их в манифесте приложения. Дальнейшие его модификации возможны только через уведомления. Плитка приложения может вернуться в исходное состояние, если отсутствуют уведомления, которые необходимо отобразить. Это возможно, если пользователь находится в автономном режиме или истёк срок действия уведомлений. Статические плитки прорисовываются поверх установленного цвета фона приложения в манифесте. Если плитка содержит прозрачные области, то в этих местах будет отображаться установленный фон.

К плиткам приложения по умолчанию есть несколько требований:

– изображение должно быть в формате .png или .jpg;
– размер стандартной плитки 150×150 пикселей;

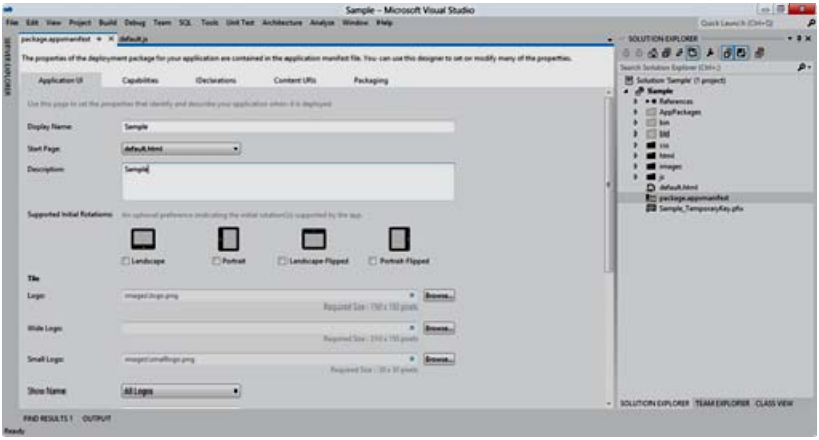

Рис. 9.5. Манифест проекта

– размер расширенной плитки 310×150 пикселей (является дополнительным);

– маленькая версия стандартной плитки 30×30 пикселей. Используется на странице поиска;

– размер файла до 150 Кб.

Для использования плиток необходимо добавить их в проект и настроить манифест проекта на их использование (рис. 9.5).

Теперь перейдём к созданию динамических плиток. Выше мы обсуждали возможные шаблоны плиток. В текущей версии Windows 8 у нас есть на выбор 30 шаблонов. Чтобы выбрать один из заготовленных шаблонов, необходимо использовать метод GetTemplateContent класса Windows.UI.Notifi cations.TileUpdateManager. Ниже приведён пример кода, который отправляет сообщение «Tile Message» на расширенную плитку нашего приложения (приложение должно содержать картинку расширенной плитки):

```
XmlDocument tileXml = TileUpdateManager.
GetTemplateContent(TileTemplateType.TileWideText04);
XmlNodeList textElements = tileXml.GetElementsByTagName("text");
textElements.Item(0).AppendChild(tileXml.CreateTextNode("Tile Message"));
TileNotifi cation tile = new TileNotifi cation(tileXml);
TileUpdateManager.CreateTileUpdaterForApplication().Update(tile);
```

Параметр метода GetTemplateContent указывает на то, какой шаблон мы хотим использовать. Каждый шаблон представлен в виде разметки xml. Так, разметка нашего шаблона следующая:

```
<?xml version="1.0" encoding="utf-16"?><tile>
<visual lang="en-US">
<binding template="TileWideText04">
<text id="1"></text>
```

```
</binding>
</visual>
</tile>
```

После добавления необходимого сообщения шаблон будет отображаться следующим образом:

```
<?xml version="1.0" encoding="utf-16"?><tile>
<visual lang="en-US">
<binding template="TileWideText04">
<text id="1">Tile Message</text>
</binding>
</visual>
</tile>
```

При помощи последней строки кода из приведённого выше примера обновляется плитка приложения. Выбранный нами шаблон позволяет отображать указанное сообщение на расширенной плитке приложения.

Рассмотренные выше возможности по изменению плитки приложения можно выполнять как локально (вышеприведённый пример), так и с сервиса, который находится в «облаке» и связан с приложением.

Помимо основной плитки приложение может иметь дополнительную. Дополнительная плитка позволяет пользователю создавать ещё один интерактивный элемент на стартовом экране Windows 8, а также открыть приложение на странице, на которой была создана дополнительная плитка.

Дополнительную плитку могут создавать только пользователи, само приложение не может её создать. При этом пользователю необходимо подтвердить, что он действительно желает создать дополнительную плитку приложения. Эта плитка даёт возможность пользователю быстро попасть в часто используемую часть приложения. Это может быть страница с часто обновляемым контентом или страница, находящаяся глубоко внутри приложения. Примеры использования дополнительной плитки:

– быстрый переход на страницу важного контакта в социальных приложениях;
– лента для программы RSS Reader;
– пользовательские события в приложении Календарь и т.д.

Ситуация, в которой пользователю необходимо иметь быстрый доступ к часто обновляемому контенту или странице в глубине навигации приложения, является отличной возможностью для использования дополнительной плитки. Для применения дополнительной плитки хорошо подходят следующие сценарии:

– отображение последних новостей из RSS;
– данные о погоде;
– обновление статусов друзей в социальных сетях;
– журнал событий из календаря пользователя.

Любой часто изменяющийся контент нашего приложения может быть размещён в дополнительной плитке. Требования к дополнительной плитке приложения преимущественно совпадают с требованиями к основной:

– изображение размером 150×150 пикселей;

– использование шаблонов плитки для определения внутреннего контента;

– может содержать расширенный логотип размером 310×150 пикселей;

– автоматическое удаление при удалении приложения;

– размещение как на рабочем столе, так и при просмотре списка всех приложений;

– может отображать уведомления и значки.

Хотя у основной и дополнительной плитки много общего, есть и ряд отличий:

– дополнительная плитка создаётся пользователем во время выполнения приложения;

– при добавлении плитки пользователь должен подтвердить данное действие;

– дополнительную плитку пользователь может удалить в любой момент без удаления программы.

Для добавления плитки необходимо создать объект класса Windows.UI.StartScreen.SecondaryTile и указать минимальный набор данных:

```
var secondaryTile = new Windows.UI.StartScreen.SecondaryTile();
secondaryTile.TileId = "3gFDG34GFS456FER";
secondaryTile.ShortName = "2nd Tile";
secondaryTile.DisplayName = "Secondary Tile";
secondaryTile.Arguments = "Some arguments";
secondaryTile.Logo = new Uri("ms-resource:Assets/secondLogo.png");
secondaryTile.WideLogo = new Uri("ms-resource:Assets/secondWideLogo.png");
```

Для объекта SecondaryTile обязательно должны быть инициализированы следующие свойства:

– TileId – идентификатор плитки. С помощью этого значения можно в дальнейшем получать доступ к плитке и обновлять её содержимое. Статический метод FindAllAsync класса SecondaryTile позволяет получить доступ к плитке, а метод экземпляра UpdateAsync – обновить его;

– ShortName – сокращённое имя, которое отображается непосредственно на плитке;

– DisplayName – полное название приложения, которое отображается в подсказке плитки, в списке всех установленных приложений и в панели управления;

– Arguments – параметры, которые передаются в приложение при запуске с дополнительной плитки;

– Logo – логотип приложения для дополнительной плитки;

– WideLogo – расширенный логотип для дополнительной плитки. Свойство WideLogo необходимо инициализировать, только если нужно использовать расширенную дополнительную плитку для нашего приложения. После инициализации всех обязательных данных необходимо вызвать метод RequestCreateAsync экземпляра класса SecondaryTile: secondaryTile.RequestCreateAsync();

Этот метод может принимать объект типа Point, который указывает координаты отображения диалога (рис. 9.6) о подтверждении размещения дополнительной плитки приложения. Можно также использовать метод RequestCreateForSelectionAsync, который позволяет более гибко управлять местоположением и размерами диалога подтверждения.

Рис. 9.6. Реализация объекта типа Point

Кроме обязательных данных, которые мы инициализировали, есть ряд параметров для дополнительной плитки:

```
secondaryTile.BackgroundColor = "Yellow";
secondaryTile.ForegroundText =
Windows.UI.StartScreen.ForegroundText.Dark;
secondaryTile.TileOptions =
Windows.UI.StartScreen.TileOptions.ShowNameOnLogo;
secondaryTile.SmallLogo = new Uri("ms-resource:Assets/secondSmallLogo.png");
```

– BackgroundColor – фоновый цвет дополнительной плитки;

– ForegroundText – цвет текста плитки;

– TileOptions – настройка отображения сокращённого имени на дополнительной плитке;

– SmallLogo – логотип дополнительной плитки размером 30×30 пикселей.

Каждое приложение может иметь несколько таких плиток. При этом необходимо помнить, что пользователь решает – размещать дополнительные плитки приложения или нет. У нас же остаётся возможность решать, какие части приложения пользователь может использовать как дополнительные плитки.

Во время разработки функционала по добавлению дополнительной плитки желательно следовать ряду рекомендаций. А именно, кнопку добавления удаления плитки размещать во встроенном меню приложения и использовать стандартные значки (рис. 9.7) для этой кнопки. Если пользователь находится на странице приложения, ссылка на которую не может быть размещена как дополнительная плитка, кнопку добавления отображать не следует. В отдельных сценариях возможны варианты размещения кнопки добавления/удаления плитки вне меню приложения. В таких случаях кнопка не должна дублироваться в меню приложения или должна быть неактивна.

При разработке логики размещения дополнительной плитки следует подумать об алгоритме формирования идентификаторов для плиток. Это поможет решить, что с ними делать при повторной установке приложения на другом устройстве пользователя.

Рис. 9.7. Стандартные значки для плитки

Не следует применять дополнительную плитку приложения как ссылку на статический контент или как команду для запуска программы.

Контракты приложения

Еще одной возможностью интеграции приложения с платформой Windows 8 является использование контрактов. Контракты представляют собой соглашение между Windows 8 и одним или несколькими приложениями для Windows Store.

Для каждого из возможных вариантов интеграции приложения с платформой существует свой контракт. Например, Windows 8 позволяет обмениваться данными между приложениями. Приложение, которое является источником таких данных, поддерживает для этого соответствующий контракт и должно удовлетворять его требованиям. Аналогично для приложений, которые являются потребителями таких данных. Они должны быть интегрированы с соответствующим контрактом, который позволит им получать данные.

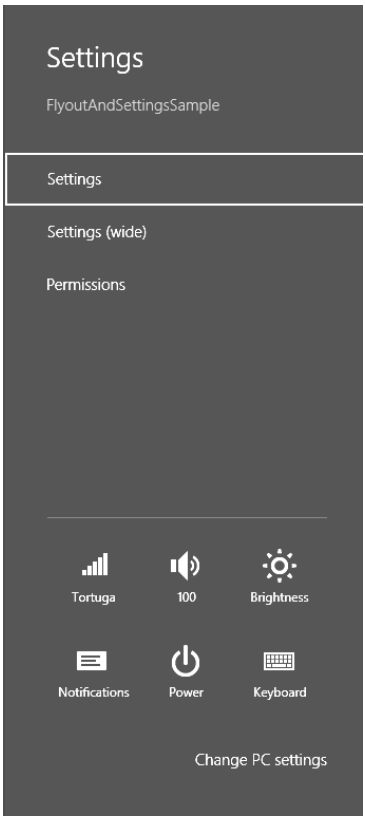

Рис. 9.8. Контракты в панели настроек

Когда приложение использует контракты такого рода, они обязательно должны быть указаны в списке возможностей. В текущей версии платформы мы можем использовать следующие контракты:

- Settings;
- Search;
- Sharing;
- Play To;
- App to App Picking.

Рассмотрим первые три контракта.

Настройки приложения (*Settings*)

При использовании этого типа контакта приложение может размещать свои элементы в системной панели настроек (рис. 9.8).

Панель настроек можно отобразить, используя возможности платформы или из приложения. Для открытия панели из приложения необходимо вызвать статический метод Show() класса Windows.UI.ApplicationSettings.SettingsPane.

Для добавления элементов списка в системную панель настроек и выполнения необходимой логики по его нажатию необходимо выполнить следующие действия:

```
var settingsPane = Windows.UI.ApplicationSettings.SettingsPane
.GetForCurrentView();
settingsPane.CommandsRequested += settingsPane_CommandsRequested;
void settingsPane_CommandsRequested(SettingsPane sender,
SettingsPaneCommandsRequestedEventArgs args)
{
args.Request.ApplicationCommands.Add(new
SettingsCommand("SomeId", "Log Out", (handler)=>
{
}));
}
```

Метод GetForCurrentView возвращает объект типа SettingsPane, который, в свою очередь, позволяет подписаться на событие CommandsRequested и в обработчике события добавить необходимые команды в системную панель настроек. При добавлении элементов списка в эту панель необходимо создать объект класса SettingsCommand и указать в конструкторе идентификатор элемента, текстовую строку для отображения в интерфейсе и делегат, который будет обрабатывать нажатия этого элемента списка.

Элементы, созданные в панели настроек, будут находиться на ней, только когда пользователь работает с приложением.

Интеграция с поиском

Интеграция с поиском позволяет приложению взаимодействовать с системной панелью поиска. Таким образом, пользователь может использовать один и тот же интерфейс для поиска контента как в операционной системе Windows 8, так и внутри приложений. Кроме того, пользователь может использовать один и тот же поисковый запрос между различными приложениями и выполнять поиск внутри приложения из любого места.

Используя данный тип контракта, мы соглашаемся с тем, что контент приложения доступен для механизма поиска платформы. Перед использованием этого контракта следует указать в манифесте, что приложение его использует (рис. 9.9).

Дальнейшая интеграция с поиском происходит через объект класса Windows. ApplicationModel.Search.SearchPane. Чтобы получить к нему доступ, следует вызвать статический метод GetForCurrentView() этого же класса. Ключевые события этого класса:

– QuerySubmitted – уведомляет о том, что приложению следует выполнить поиск по указанному поисковому запросу от пользователя;

– SuggestionsRequested – позволяет приложению предоставить на выбор пользователя список подсказок для запроса;

– VisibilityChanged – уведомляет об открытии/закрытии панели поиска.

Совместное использование данных (*Sharing*)

Возможность совместного использования данных позволяет пользователям работать с несколькими приложениями или службами одновременно. При использовании этого контракта не нужно писать дополни-

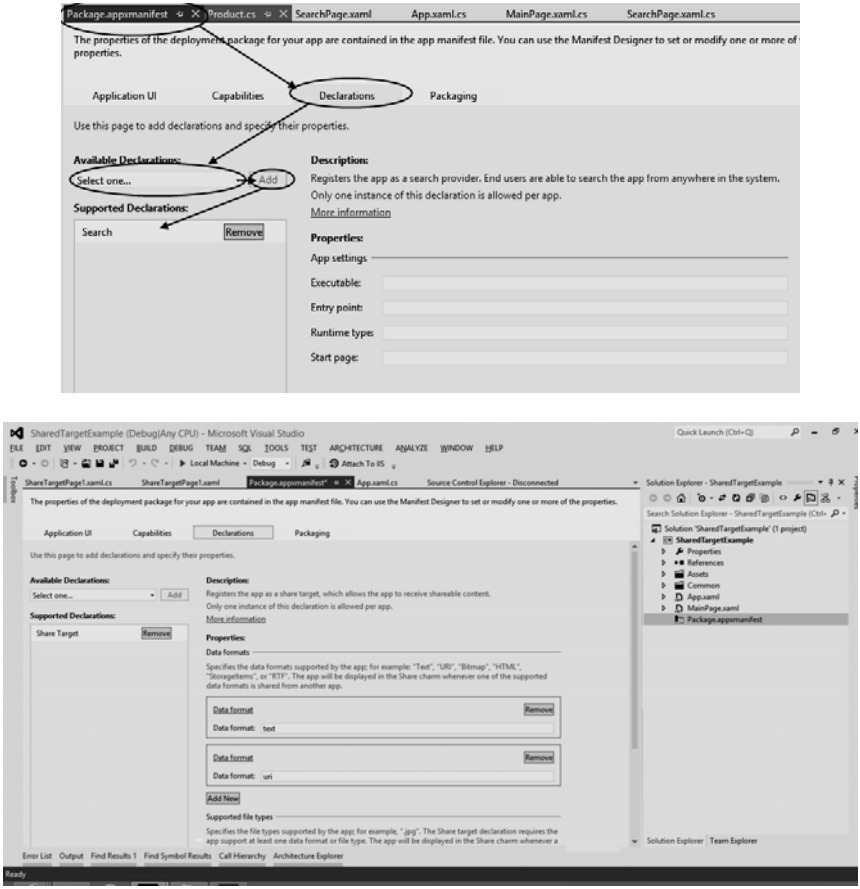

Рис. 9.9. Настройка контракта в манифесте

тельный код или руководство для других разработчиков только для того, чтобы реализовать механизм совместного использования данных.

Во время работы с этим видом контракта различают два вида приложений:

– приложения, которые является источниками данных и отдают их в различных форматах;

– приложения, которые используют данные совместно.

Начнём с механизма, который позволяет предоставлять данные для других приложений/сервисов.

В основе механизма обмена данными лежит объект класса Windows. ApplicationModel.DataTransfer.DataTransferManager. Доступ к нему можно получить, вызвав статический метод GetForCurrentView() этого же класса:

```
datatransferManager = DataTransferManager.GetForCurrentView();
datatransferManager.DataRequested += new TypedEventHandler
<DataTransferManager, DataRequestedEventArgs>(this.DataRequested);
```

В обработчике события DataRequested можно указать данные, которые необходимо предоставить другим приложениям. Пример:

```
void DataRequested(DataTransferManager sender, DataRequestedEventArgs e)
{
e.Request.Data.Properties.Title = this.dataPackageTitle;
e.Request.Data.Properties.Description =
this.dataPackageDescription;
if (this.dataPackageThumbnail != null)
{
e.Request.Data.Properties.Thumbnail =
this.dataPackageThumbnail;
}
e.Request.Data.SetText(this.dataPackageText);
}
```

Через свойство Properties можно указать описание данных для совместного использования.

Для приложения, которое использует совместные данные, необходимо переопределить метод OnShareTargetActivated класса Application и через параметры получить доступ к совместным данным:

```
protected override async void OnShareTargetActivated
(ShareTargetActivatedEventArgs args)
{
if (args.Kind == Windows.ApplicationModel.
Activation.ActivationKind.ShareTarget)
{
var text = await args.ShareOperation.Data.
GetTextAsync()
}
}
```

Кроме текста, приложения могут обмениваться файлами, изображениями, контентом html и др. Можно определять свой собственный формат и передавать его другим приложениям. При этом важно, чтобы приложение-потребитель знало формат принимаемых данных.

10. ТЕСТИРОВАНИЕ И ОТЛАДКА ПРИЛОЖЕНИЙ ДЛЯ WINDOWS STORE

Независимо от идеи, реализуемой в приложении, пользователи вряд ли получат удовольствие от неработоспособного и медленно работающего продукта. Поэтому в процессе разработки приложений применяются различные механизмы, позволяющие улучшить работоспособность продукта и его производительность.

Тестирование позволяет повысить качество работы нашего приложения. Во время тестирования можно применять нескольких методик тести-

Рис. 10.1. Боковая панель симулятора

Рис. 10.2. Режим ориентации симулятора

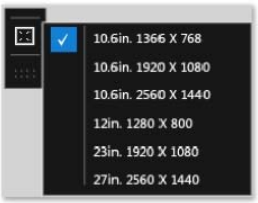

Рис. 10.3. Выбор различных разрешений

рования программного обеспечения одновременно. Каждая из используемых методологий позволяет выявить определённого рода ошибки в работе нашего приложения. Кроме тестирования, следует уделить достаточно внимания улучшению производительности нашего приложения. Так как медленное приложение вряд ли доставит удовольствие пользователям.

Отладка

В первой главе мы уже познакомились немного с симулятором устройства с Windows 8. Теперь давайте более детально рассмотрим его возможности, а именно боковую панель (рис. 10.1).

Первый блок из четырёх кнопок позволяет выбрать режим взаимодействия с интерфейсом симулятора:

– курсором мыши;

– эмуляцией касания одним пальцем.

– «Курсор» в рамках эмулятора приобретает форму

– эмуляция мультисенсорного управления.

– «Курсор» в рамках эмулятора приобретает форму:

Чтобы работало мультисенсорное управление, необходимо зажать левую кнопку мышки и крутить колесо мыши. Обычные касания в этом режиме не работают эмуляция вращения. Форма «курсора» аналогична предыдущему режиму. Отличием является только поведение при прокрутке колёсиком мышки.

Следующий блок из двух кнопок позволяет задать режим ориентации симулятора (рис. 10.2).

Симулятор Windows 8 позволяет тестировать приложения на шести различных разрешениях экрана. Чтобы их изменять, используйте следующую кнопку (рис. 10.3).

Кнопка для эмуляции географических координат является следующей в списке и открывает окно для задания координат (рис. 10.4).

Кнопка (рис. 10.5), расположенная ниже, позволяет получить снимок экрана симулятора и сохранить его либо в оперативной памяти, либо в файл. Это зависит от настроек, которые мы указываем в диалоговом окне настроек симулятора. Окна настроек симулятора открывается следующей кнопкой (рис. 10.6).

Последняя кнопка в списке открывает веб-страницу помощи по работе с симулятором устройства с Windows 8.

Тестирование

Как упоминалось выше, во время тестирования применяются различные виды тестов. Дополнительную информацию о видах тестирования см. в соответствующей литературе. Рассмотрим создание юнит-тестов для проекта.

Создание юнит-тестов

Visual Studio 2011 Beta содержит соответствующий шаблон для создания юнит-тестов (рис. 10.7).

Рис. 10.4. Эмуляция географических координат

Рис. 10.5. Снимок экрана симулятора

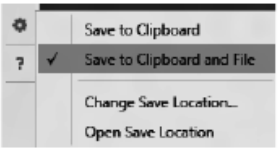

Рис. 10.6. Сохранение результатов

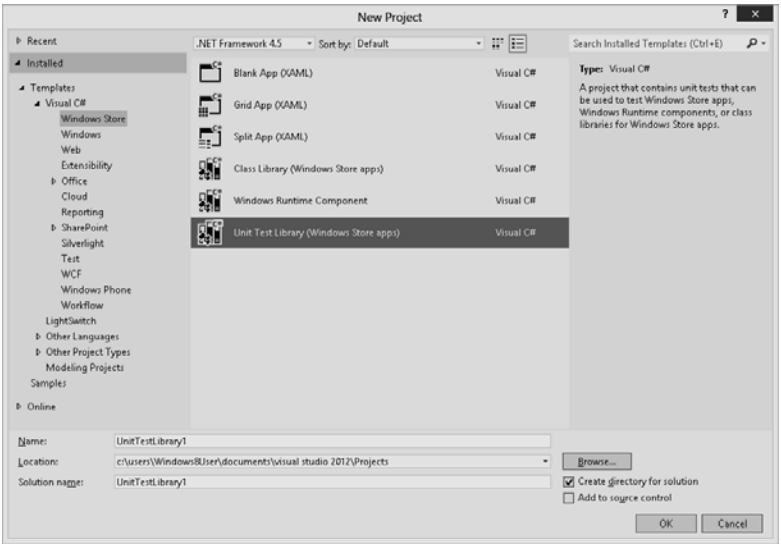

Рис. 10.7. Шаблон создания тестов

После создание проекта для юнит-тестов модифицируем тест по умолчанию, чтобы он имел следующий вид:

```
[TestClass]
public class UnitTest1
{
[TestMethod]
public void TestMethod1()
{
Assert.AreEqual(string.Empty, «»);
}
[TestMethod]
public void TestMethod2()
{
Assert.Fail();
}
[TestMethod]
public void TestMethod3()
{
Assert.Inconclusive();
}
}
```

Для написания юнит-тестов используются всё те же методы класса Assert, но в данном курсе они рассматриваться не будут.

Итак, есть готовые юнит-тесты для приложения. Для их запуска понадобится окно Unit Test Explorer. Его можно открыть через поле ввода Quick Launch (находится в правом верхнем углу Visual Studio 2011) или выбрать в главном меню Unit Test->Windows->Unit Test Explorer.

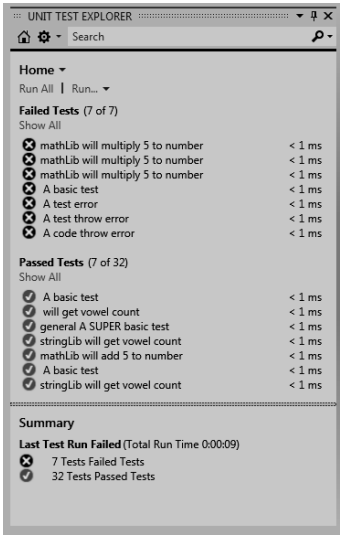

Рис. 10.8. Результат работы тестов

В окне Unit Test Explorer мы можем запустить все тесты нашего приложения или выбрать определённую группу тестов, которая нас интересует. После работы наших юнит-тестов мы увидим результат их работы (рис. 10.8.)

Повышение производительности приложений

Для того чтобы понять, в каких местах или при каких сценариях производительность приложения низкая, необходимо использовать соответствующие инструменты, которые позволяют получить информацию об использовании ресурсов устройства вашим прило-

жением. Такие инструменты называются профайлерами, и Visual Studio 2011 Express Beta содержит в своём составе.

Профилирование приложений для Windows Store

Инструменты профилирования приложений для Windows Store в Visual Studio 2012 Express Beta позволяют измерять значения и определять ключевые моменты работы наших приложений. Профайлер собирает временную информацию приложений, написанных на Visual C++, Visual C# и Visual Basic, используя соответствующие методы, которые вызываются через определённый интервал в соответствии со стеком вызовов ЦПУ. Графическое отображение результатов профилирования легко позволяет найти нужное место в приложении и определить производительность его отдельных частей. У нас есть возможность профилировать приложение от момента его запуска и до остановки или же запускать профилирование только в интересующих нас сценариях.

Во время профилирования приложения желательно выбирать для него тип сборки Release, потому что именно в этом типе сборки приложение по своему содержимому больше всего похоже на то, которые пользователь поставит себе из Microsoft Store.

Для запуска профилирования приложения выбрать в главном меню Visual Studio пункт Debug->Start Performance Analysis или нажать комбинацию клавиш Alt+F12 (рис. 10.9).

После выполнения необходимых сценариев работы приложения следует остановить анализ производительность приложения, и приложение Visual Studio начнёт формирование отчёта производительности приложения. Результатом будет следующий отчёт (рис. 10.10).

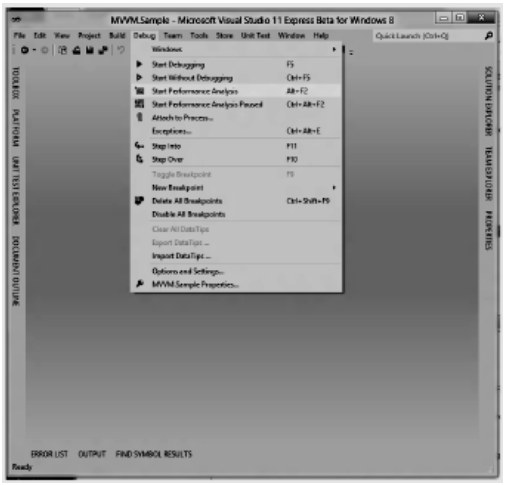

Рис. 10.9. Профилирование приложения

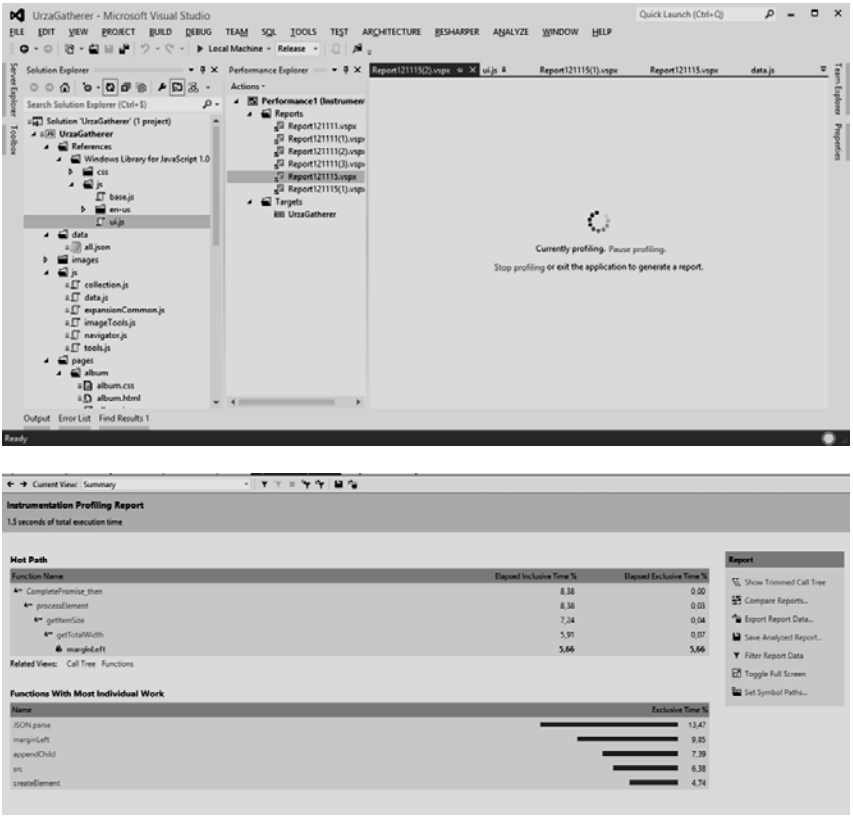

Рис. 10.10. Отчёт о приозводительности приложения

Другой механизм отслеживания работы приложения – это включение счётчиков кадров. В текущей версии это делается путём изменения ключей в регистре:

– *для 32-битной Windows 8:*

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Xaml] «EnableFrameRateCounter»=dword:00000001

– *для 64-битной Windows 8:*

[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Xaml] "EnableFrameRateCounter"=dword:00000001

После изменения соответствующего ключа в реестре системы при запуске приложения будут отображаться счётчики кадров (рис 10.11).

Рис. 10.11. Счётчик кадров

Первое значение слева показывает количество кадров, обрабатываемых в секунду в потоке компоновщика, т.е. на GPU видеокарты. Второе число – количество кадров в секунду, которое обрабатывает центральный процессор. Третье – использование памяти приложением. Последние два значения отображают время в миллисекундах для потока компоновщика и интерфейсного потока соответственно.

Таким образом, используя рассмотренные выше способы контроля работоспособности и анализа производительности приложения, можно стабилизовать его работу и предоставить пользователям продукт высокого качества.

ЗАКЛЮЧЕНИЕ

В учебном пособии рассмотрены основы разработки информационных систем для операционной системы Windows 8 RT, такие как жизненный цикл программной системы, проектирование интерфейса, реализация приложения. Данные системы являются наиболее динамично развивающимися и используются конечным пользователем в повседневной жизни.

Материалы, изложенные в работе, позволяют не только понять сущность процесса разработки программного обеспечения, но и использовать в реальных проектах.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. **Мейер, Б.** Учебный курс «Основы объектно-ориентированного программирования». http://www.intuit.ru/department/se/oopbases/

2. **Основы** разработки программного обеспечения вычислительных систем : учебное пособие / В. И. лоскутов, И. В. милованов. – Тамбов : Изд-во ГОУ ВПО ТГТУ, 2011. – 80 с.

3. **Приёмы** объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – Москва : ДМК пресс, 2007, 368 с.

4. **Сергеев, С. Ф.** Введение в проектирование интеллектуальных интерфейсов : учебное пособие / С. Ф. Сергеев, П. И. Падерно, Н. А. Назаренко. – Санкт-Петербург : СПбГУ ИТМО, 2011. – 108 с.

5. **Буч, Г.** Язык UML : пер. с англ. / Г. Буч, Д. Рамбо, А. Джекобсон. – Москва : ДМК, 2000.

6. **Разработка** приложений для Windows Store. http://msdn. microsoft.com/ru-ru/library/windows/apps/xaml/BR229566#

7. **Центр** разработчиков Windows. Научитесь создавать приложения в стиле «Metro». http://msdn.microsoft.com/ruru /library/windows/ apps/br229519.aspx

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ ……. 3

ВВЕДЕНИЕ ……. 3

1. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ОСНОВНЫЕ ЭТАПЫ СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С АРХИТЕКТУРОЙ WINRT …… 5

2. ИНТЕРФЕЙС ПРИЛОЖЕНИЙ ДЛЯ WINDOWS STORE. ТРЕБОВАНИЯ К ВНЕШНЕМУ ВИДУ ПРИЛОЖЕНИЯ, ИНТЕРФЕЙС ПРИЛОЖЕНИЯ, ЭЛЕМЕНТЫ УПРАВЛЕНИЯ …….. 9

3. ПРОЦЕСС ПРОЕКТИРОВАНИЯ ПРИЛОЖЕНИЙ ДЛЯ WINDOWS STORE ……. 13

4. РАЗРАБОТКА ГИБРИДНЫХ ПОДСИСТЕМ. ИСПОЛЬЗОВАНИЕ НЕСКОЛЬКИХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ В ОДНОМ ПРИЛОЖЕНИИ …….. 19

5. АРХИТЕКТУРА ПРИЛОЖЕНИЯ ДЛЯ WINDOWS STORE ……. 24

6. ЖИЗНЕННЫЙ ЦИКЛ ПРИЛОЖЕНИЯ С АРХИТЕКТУРОЙ WINRT. ОСОБЕННОСТИ АРХИТЕКТУРЫ. ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТАРИЯ CLR, .NET. МЕТОДЫ ПРОГРАММИРОВАНИЯ 30

7. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА METRO. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ. СТИЛИ. РЕСУРСЫ ……. 36

8. ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННОЙ СИСТЕМЫ ДЛЯ WINDOWS STORE. ОБРАБОТКА ДАННЫХ ……. 46

9. РАЗРАБОТКА И РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ ДЛЯ WINDOWS STORE …….. 59

10. ТЕСТИРОВАНИЕ И ОТЛАДКА ПРИЛОЖЕНИЙ ДЛЯ WINDOWS STORE ……. 71

ЗАКЛЮЧЕНИЕ ……... 77

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ ……... 78

Учебное издание

ЛОСКУТОВ Вячеслав Иванович,
КОРОБОВА Ирина Львовна

РАЗРАБОТКА ИНФОРМАЦИОННЫХ СИСТЕМ ДЛЯ WINDOWS STORE

Учебное пособие

Редактор Л. В. Комбарова
Инженер по компьютерному макетированию И. В. Евсеева

ISBN 978-5-8265-1285-2

Подписано в печать 26.06.2014.
Формат 60 × 84 / 16. 4,65 усл. печ. л.
Тираж 100 экз. Заказ № 318

Издательско-полиграфический центр
ФГБОУ ВПО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел. 8(4752) 63-81-08
E-mail: izdatelstvo@admin.tstu.ru