А. В. Абрамян, М. Э. Абрамян

РАЗРАБОТКА ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА НА ОСНОВЕ ТЕХНОЛОГИИ WINDOWS PRESENTATION FOUNDATION

Михаил Эдуардович Абрамян
Анна Владимировна Абрамян

Разработка пользовательского интерфейса на основе технологии Windows Presentation Foundation

http://www.litres.ru/pages/biblio_book/?art=39846707
Разработка пользовательского интерфейса на основе системы Windows Presentation Foundation:
ISBN 9785927523757

Аннотация

В учебнике рассмотрены основные приемы разработки пользовательского интерфейса на основе технологии Windows Presentation Foundation (WPF), входящей в состав платформы .NET, начиная с версии 3.0. Учебный материал излагается в форме подробного описания 19 проектов для среды программирования Microsoft Visual Studio 2015, демонстрирующих различные аспекты технологии WPF. Описание проектов сопровождается многочисленными комментариями. Завершающий раздел содержит 48 учебных заданий, предназначенных для закрепления изученного

материала. Для студентов бакалавриата, обучающихся по направлению подготовки 02.03.02 «Фундаментальная информатика и информационные технологии».

Содержание

А. В. Абрамян, М. Э. Абрамян Разработка пользовательского интерфейса на основе системы Windows Presentation Foundation

Предисловие

Книга знакомит читателя с основными приемами разработки пользовательского интерфейса на основе системы Windows Presentation Foundation (WPF), входящей в платформу .NET, начиная с версии 3.0. Данная система была создана с целью снять ряд серьезных ограничений, имевшихся у ее предшественницы – системы разработки интерфейса Windows Forms, изначально входившей в состав платформы .NET. При этом наряду с сохранением концепций, лежащих в основе Windows Forms (в частности, механизма

обработки событий), система WPF была дополнена новыми технологиями, позволяющими разрабатывать интерфейсы, имеющие существенно более широкие графические возможности и, что не менее важно в современном мире планшетов и смартфонов, допускающими автоматическую адаптацию к особенностям устройств, на которых запущено приложение.

Новой важной чертой системы WPF для разработчиков приложений стало явное разграничение программного кода и макета приложения, которое было обеспечено включением в состав проекта XML-файлов, позволяющих описывать визуальный интерфейс на особом декларативном языке разметки XAML (eXtensible Application Markup Language). Следует отметить, что в настоящее время подобный подход реализован в большинстве систем разработки интерфейсов.

Разработчики приложений на основе технологии WPF получили в свое распоряжение новые концепции, основанные на свойствах зависимостей, маршрутизируемых событиях и привязке данных. Созданные в рамках WPF иерархии классов и визуальных компонентов, в том числе компонентов с содержимым и группирующих компонентов, обеспечивающих динамическую компоновку своих дочерних элементов, позволили гибко комбинировать интерфейсные элементы, обеспечивая подходящее визуальное представление на экране любого размера.

Принципиально новой по сравнению с Windows Forms стала графическая подсистема, основанная на технологии

DirectX и позволяющая реализовывать качественную и быструю двумерную и трехмерную графику и анимацию. Был осуществлен переход на аппаратно-независимую систему единиц измерения (вместо экранных пикселов), обеспечивший одинаковый внешний вид приложения на экране с любым разрешением.

Не последнее место в списке преимуществ технологии WPF занимает удобство разработки WPF-приложений, которое обеспечивается средствами среды программирования Microsoft Visual Studio (версии 2008–2015). Традиционные методы визуальной разработки интерфейса, имевшиеся уже в технологии Windows Forms, были дополнены методами, основанными на непосредственном редактировании xaml-файлов, определяющих макет графического приложения. Подобное редактирование (как и редактирование программного кода) существенно упрощается благодаря контекстным подсказкам, автоматической проверке синтаксиса, средствам автозавершения и другим средствам, встроенным в редакторы кода и xaml-файлов.

Учитывая перечисленные выше особенности системы WPF и принимая во внимание широкую распространенность технологий и языков программирования, основанных на платформе .NET, представляется вполне оправданным применение данной системы в качестве базовой при изучении приемов и методов разработки пользовательского интерфейса в рамках соответствующего университетского кур-

са.

При этом, однако, возникают две проблемы. Первая обусловлена сложностью системы WPF и обширностью ее средств, которые невозможно охватить в рамках одного курса. Вторая проблема связана с недостатком учебной литературы. Если ограничиться изданиями, переведенными на русский язык, то можно отметить лишь [7–9] (при включении в рассмотрение английских книг их общее количество увеличится лишь в 2–3 раза). Из этих изданий на роль учебника может отчасти претендовать только книга Чарльза Петцольда [7], написанная с большим методическим мастерством (что характерно для всех работ этого автора) и содержащая много примеров законченных программ. Две другие книги являются, скорее, справочниками, включающими небольшие фрагменты иллюстративного кода.

Предлагаемая книга представляет собой попытку решения обеих проблем. С одной стороны, она посвящена лишь основным возможностям технологии WPF, которые вполне можно освоить за семестровый курс, а с другой – излагает материал в «практической» форме, упрощающей его усвоение. Изложение построено в виде подробного описания ряда законченных программных проектов, каждый из которых посвящен одному из аспектов технологии WPF. Многочисленные комментарии содержат дополнительные сведения, связанные с изучаемыми возможностями. Во многих случаях приводится сравнение рассматриваемых средств WPF

с аналогичными средствами библиотеки Windows Forms. В процессе разработки проектов авторы намеренно допускают типичные ошибки, характерные для начинающих разработчиков интерфейсов, подробно объясняют их причины и приводят способы исправления. Подобный вариант изложения учебного материала «на примерах» ранее с успехом применялся авторами при чтении ими курсов по разработке пользовательских интерфейсов с использованием библиотеки VCL системы программирования Borland Delphi [1, 2] и предшественницы WPF – библиотеки Windows Forms платформы .NET [3].

Учебник состоит из описаний 19 проектов. Его содержимое можно разбить на три части. В первой части рассматриваются базовые возможности библиотеки WPF: создание проекта для WPF-приложения, работа с xaml-файлами, использование группирующих компонентов, управление программой посредством обработчиков событий (проект EVENTS), приемы работы с окнами в WPF-приложении, организация взаимодействия между окнами, особенности диалоговых окон (проект WINDOWS), совместное использование обработчиков событий, события клавиатуры (проект CALC), таймеры в WPF-приложении (проект CLOCK), возможности полей ввода, организация проверки правильности введенных данных (проект TEXTBOXES), события мыши (проект MOUSE), механизм перетаскивания Drag & Drop (проект ZOO), работа с курсорами и иконками,

создание ресурсов приложения, совместное использование средств WPF и Windows Forms (проект CURSORS).

Вторая часть содержит описание разработки одного большого проекта TEXTEDIT, разбитое на 5 этапов (версий проекта). В ней основное внимание уделяется особенностям тех стандартных интерфейсных элементов, без которых не обходится практически ни одно приложение: меню и различных видов его команд (версии 1 и 2), контекстных меню (версия 3), панели инструментов (версия 4), статусной панели (версия 5). Кроме того, в версии 1 подробно рассматриваются особенности организации работы с файлами (открытие, сохранение, контроль за сохранением внесенных изменений), а также описываются приемы работы с командами WPF; в версии 2 рассказывается о том, как создавать новые команды WPF; в версии 3 рассматриваются особенности реализации команд редактирования; а в версиях 4 и 5 особое внимание уделяется различным аспектам механизма привязки компонентов и, кроме того, описываются действия по созданию новых свойств зависимости.

Третья часть содержит проекты, связанные с дополнительными возможностями WPF – работа с цветами и кистями и использование компонентов TrackBar (проект COLORS), работа со списками и использование стилей при определении макета приложения (проект LISTBOXES), работа с флажками и наборами флажков (проект CHECKBOXES), работа с иерархическими списками,

реализация дерева каталогов и списка файлов, использование компонентов GridSplitter и Image, применение реестра Windows для хранения настроек приложения (проект IMGVIEW), работа с табличными списками и использование заставок и градиентных кистей (проект TRIGFUNC), создание компонентов во время выполнения программы (проект HTOWERS).

Завершает книгу раздел с 48 учебными заданиями, разбитыми на 5 групп. Каждая группа содержит однотипные задания; первая группа включает задания на организацию взаимодействия между окнами приложения, вторая – на синхронизацию компонентов и совместное использование обработчиков событий, третья – на реализацию режима перетаскивания Drag & Drop, четвертая – на создание программ, управляемых таймером, пятая – на использование стандартных диалоговых окон и работу с реестром. Большое количество заданий позволяет легко формировать из них наборы индивидуальных заданий одинакового уровня сложности.

За рамками настоящей книги осталось большинство возможностей библиотеки WPF, связанных с ее графической подсистемой, поскольку детальное изучение этих возможностей более естественно отнести к курсу по компьютерной графике.

Предполагается, что читатель книги уже изучил базовый курс по программированию. Знание основ языка C#, а также языка XML, является желательным, но не обязательным, так

как ознакомиться с конструкциями этих языков можно в процессе чтения книги. Тем не менее полезной может оказаться книга [5], освещающая практически все аспекты языка C# и стандартных библиотек платформы .NET, а также книга [4], содержащая описание основных типов стандартной библиотеки (в том числе классов, связанных с обработкой строк и файлов), объектной модели языка C# и технологии LINQ, в том числе интерфейса LINQ to XML.

В качестве среды программирования используется Microsoft Visual Studio 2015, однако все проекты можно реализовать и в более ранних версиях этой среды (начиная с версии 2008). Применение новых возможностей языка C#, появившихся в его версии 6.0 и доступных только в Visual Studio версий 2015 и выше, всегда особо отмечается и сопровождается альтернативными вариантами кода для предыдущих версий.

Соглашения по оформлению фрагментов программного кода и xaml-файлов приводятся в первом проекте EVENTS. В нем же описываются основные действия по созданию и редактированию проекта для WPF-приложения.

1. События: EVENTS

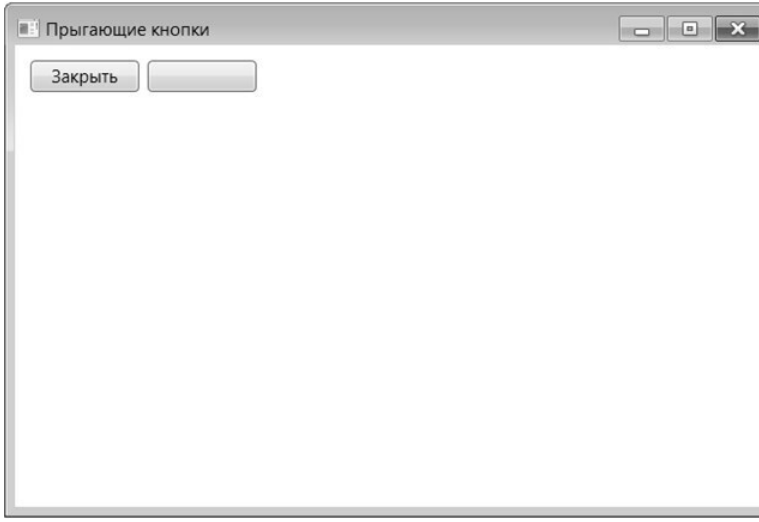

Рис. 1. Окно приложения EVENTS

1.1. Создание проекта для WPF-приложения

Для того чтобы создать проект в среде программирования Visual Studio, выполните команду File | New | Project (Ctrl +Shift+N), в появившемся окне New Project выберите в левой части вариант Visual C#, а в правой части – вариант WPF Application, в поле ввода Name укажите имя проекта (в нашем случае EVENTS), а в поле ввода Location укажите каталог, в котором будет создан каталог проекта. Желательно снять флажок Create directory for solution, чтобы не создавался промежуточный каталог для решения (каталог для решения удобно использовать в ситуации, когда решение содержит *несколько* проектов; в нашем случае решение всегда будет содержать единственный проект). После указания всех настроек нажмите кнопку «OK».

В результате будет создан каталог EVENTS, содержащий все файлы одноименного проекта, в том числе файл решения EVENTS.sln, файл проекта EVENTS.csproj, а также файлы для двух основных классов проекта, созданных автоматически: класса MainWindow, представляющего главное окно программы, и класса App, обеспечивающего запуск программы, в ходе которого создается и отображается на экране экземпляр главного окна.

Для каждого класса создаются два файла: с расширени-

ем xaml, котрый содержит часть определения класса в специальном формате, и с расширением cs (перед которым тоже содержится текст xaml), содержащий часть определения класса на языке C#. Файл с расширением xaml (*xaml-файл*) имеет формат XML (eXtensible Markup Language – расширяемый язык разметки). Аббревиатура XAML (произносится «зэмл» или «замл») означает, что используется специализированный вариант языка XML: eXtensible Application Markup Language – расширяемый язык разметки *для приложений*.

Приведем содержимое файлов, связанных с классом App и созданных в Visual Studio 2015.

App.xaml:

```
<Application x:Class="EVENTS.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:EVENTS"
    StartupUri="MainWindow.xaml">
  <Application.Resources>

  </Application.Resources>
</Application>
```

App.xaml.cs:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace EVENTS
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
    }
}
```

Анализ этих файлов показывает, что класс App наследуется от стандартного класса Application, а также что в cs-файле никакой новой функциональности в класс App не добавляется (обратите внимание на то, что при определении класса App в cs-файле указывается модификатор partial, означающий, что часть определения этого класса содержится в другом файле). Созданный класс (как и другие классы проекта, создаваемые автоматически) связывается с пространством имен EVENTS, совпадающим с именем проекта.

Перед анализом содержимого xaml-файла следует предварительно описать основные правила, по которым формируется любой XML-файл. Подобные файлы состоят из

иерархического набора вложенных друг в друга именованных *XML-элементов*, причем каждый элемент может иметь любое количество *XML-атрибутов* и *дочерних элементов*. Элементы оформляются в виде *тегов*; открывающий тег элемента имеет вид <*имя_элемента список_атрибутов*>, а закрывающий тег – </*имя_элемента*>. Между этими тегами располагается *содержимое* элемента, которое может представлять собой обычный текст и/или другие (дочерние) элементы (а также другие *XML-узлы*, которые мы не будем обсуждать, так как в xaml-файле они не используются). Число уровней вложенности элементов может быть любым. Если элемент не имеет содержимого, то он может представляться в виде одного *комбинированного тега* вида <*имя_элемента список_атрибутов* />. Атрибуты в списке определяются следующим образом: *имя_атрибута*="*значение_атрибута*"; значение обязательно заключается в кавычки (одинарные или двойные). Все атрибуты одного элемента должны иметь *различные* имена, в то время как его дочерние элементы могут иметь совпадающие имена. Регистр в именах учитывается; имена как атрибутов, так и элементов могут содержать только буквы, цифры, символы «.» (точка), «-» (дефис) и «_» (подчеркивание) и начинаться либо с буквы, либо с символа подчеркивания. Пробелы в именах не допускаются. Перед именами элементов и атрибутов могут указываться *префиксы пространств имен*, отделяемые от собственно имени двоеточием (в файле App.xaml имеются два таких ат-

рибута: xmlns:x и xmlns:local). Любой XML-файл должен содержать *единственный* XML-элемент верхнего уровня, называемый *корневым элементом* (в файле App.xaml это элемент Application).

В той части определения класса App, которая размещается в xaml-файле, содержится единственная, но очень важная настройка – указание на класс, экземпляр которого будет создан при запуске программы. Это атрибут StartupUri элемента Application, его значение равно MainWindow.xaml. Фактически данный атрибут является *свойством* класса Application. Как и другие свойства, его можно настроить либо непосредственно в тексте xaml-файла, либо в окне свойств Properties, которое отображает доступные для редактирования свойства текущего объекта из xaml-файла (если в редакторе отображается не xaml-, а cs-файл, то окно Properties является пустым).

При указании или изменении свойств в xaml-файле очень помогает предусмотренная в редакторе xaml-файлов возможность *контекстной подсказки* при выборе значений свойств. Окно Properties удобно в том отношении, что позволяет просмотреть *все* доступные свойства текущего объекта. В xaml-файле отображаются только те свойства, значения которых отличаются от значений по умолчанию для данного объекта. Чтобы добавить в xaml-файл новое свойство, достаточно в окне Properties указать для данного свойства значение, отличное от значения по умолчанию.

В процессе компиляции программы все xaml-файлы конвертируются в специальный двоичный формат и затем обрабатываются совместно с cs-файлами проекта.

Класс App обычно не требуется редактировать. По этой причине после создания проекта в редактор не загружаются файлы, связанные с классом App.

> **Комментарий**
>
> На протяжении всей книги мы будем придерживаться следующих соглашений об отступах в текстах xaml- и cs-файлов. В xaml-файле каждый вложенный элемент набирается с *отступом в 2 пробела* относительно родительского элемента (причина столь небольшого отступа заключается в том, что глубина вложенности элементов в xaml-файлах может быть достаточно большой); если список атрибутов в открывающем теге элемента не умещается в одной строке, то он переносится на следующую строку с *отступом в 4 пробела* относительно начала открывающего тега. Для cs-файла ситуация обратная: вложенные конструкции набираются с *отступом в 4 пробела* (как в редакторе кода среды Visual Studio), а при переносе длинного оператора на новую строку используется *отступ в 2 пробела*.

Приведем файлы, связанные с классом MainWindow; именно эти файлы автоматически загружаются в редактор после создания (или открытия) проекта.

MainWindow.xaml:

```
<Window x:Class="EVENTS.MainWindow"
    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="clr-namespace:EVENTS"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
  <Grid>

  </Grid>
</Window>
```

MainWindow.xaml.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

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

В дальнейшем при выводе текста xaml-файлов мы не будем указывать атрибуты корневого элемента, предшествующие атрибуту Title, поскольку они генерируются автоматически и не требуют изменения.

Одновременно с отображением xaml-файла для класса окна на экране выводится *окно дизайнера* – визуального ре-

дактора. Следует заметить, что при разработке WPF-приложений визуальный редактор используется не так активно, как при разработке приложений, использующих библиотеку Windows Forms. Это связано с тем, что относительное расположение компонентов в окне WPF-приложения обычно не определяется явным образом, с указанием абсолютных оконных координат, а *вычисляется* по специальным правилам, связанным с особенностями тех или иных группирующих компонентов (называемых также *панелями*). Таким образом, окно дизайнера используется преимущественно для того, чтобы быстро определить, как те или иные изменения, внесенные в xaml-файл или сделанные с помощью окна свойств, повлияют на внешний вид окна. С помощью выпадающего списка можно настраивать масштаб для окна дизайнера.

Обратите внимание на то, что по умолчанию в окно приложения уже включен группирующий компонент Grid – наиболее универсальный из группирующих компонентов, позволяющий размещать свои дочерние компоненты в нескольких строках и столбцах. Кроме того, для класса MainWindow определены три свойства: Title, Height и Width. Гораздо большее число свойств приведено в окне Properties (как уже было отмечено выше, их отсутствие в xaml-файле объясняется тем, что данный файл содержит только свойства, значения которых отличаются от значений по умолчанию). При просмотре списка свойств в окне Properties можно исполь-

зовать либо режим, при котором «родственные» свойства объединяются в группы, либо режим, при котором свойства располагаются в алфавитном порядке. Кроме того, с помощью поля ввода, расположенного над списком свойств, можно выполнять фильтрацию этого списка, отображая только те свойства, в именах которых содержится указанная строка. Например, после ввода в это поле текста Height в списке свойств останутся лишь три свойства: Height, MaxHeight и MinHeight.

В файле MainWindow.xaml.cs содержится частичное определение класса MainWindow, включающее конструктор без параметров, в котором вызывается метод InitializeComponent, обеспечивающий начальную инициализацию всех компонентов окна. Все действия с компонентами можно выполнять только после их начальной инициализации, поэтому пользовательский код добавляется в конструктор *после* вызова данного метода.

> **Комментарий**
>
> Большинство XML-атрибутов в xaml-файле относятся либо к *атрибутам свойств* (и определяют соответствующие свойства объектов), либо к *атрибутам событий* (и позволяют связать события с методами-обработчиками). XML-элементы тоже можно разбить на две категории: это *элементы-объекты*, имена которых совпадают с типом определяемого объекта, и *элементы-свойства*, имеющие составные имена вида *тип.свойство*

(элементы-свойства используются в ситуации, когда свойство нельзя определить с помощью единственного атрибута).

Кроме того, для каждого типа компонентов WPF определено особое свойство, которое можно задать в xaml-файле, указав его в виде одного или нескольких *дочерних* элементов-объектов (примером такого свойства является свойство Content; в частности, в приведенном выше файле MainWindow.xaml свойство Content окна Window равно компоненту Grid). В подобной ситуации имя свойства вообще не указывается. Несколько дочерних элементов-объектов можно указывать, если определяемое свойство является свойством-коллекцией (примером такого свойства является свойство Children, имеющееся у всех *группирующих* компонентов-панелей, например, компонента Grid или используемого в следующем пункте компонента Canvas).

1.2. Добавление компонентов и настройка их свойств

Разрабатываемое нами приложение отличается от традиционных WPF-приложений тем, что мы хотим произвольным образом перемещать отдельные компоненты в пределах окна. В подобной ситуации вместо группирующего компонента Grid удобнее пользоваться компонентом Canvas. Поэтому нам необходимо изменить «внешний» компонент окна и, кроме того, добавить на новый внешний компонент кнопку Button.

Эти действия можно выполнить двумя способами: с помощью окна дизайнера, удалив в нем лишние компоненты и добавив новые путем их перетаскивания с панели компонентов Toolbox, и с помощью непосредственного редактирования xaml-файла.

Опишем первый способ.

Вначале необходимо выделить в окне дизайнера компонент Grid, щелкнув на нем мышью. То, что выделен именно компонент Grid, можно проверить по тексту xaml-файла (в котором также будет выделен элемент <Grid>) или по окну Properties (где указываются свойства выделенного компонента). После выделения компонента его надо удалить, нажав клавишу Delete. Обратите внимание на то, что в результате такого удаления элемент Window в xaml-файле будет

представлен в виде комбинированного тега <Window … />, поскольку теперь он не содержит дочерних элементов.

Затем необходимо добавить в окно компонент Canvas. Для этого надо развернуть панель Toolbox, которая обычно располагается у левой границы окна Visual Studio в свернутом состоянии. Если данная панель отсутствует, то ее можно отобразить с помощью команды меню View | Toolbox (Ctrl +W, X). Для быстрого поиска нужного компонента на панели Toolbox достаточно ввести начальную часть его имени в поле ввода, расположенное в верхней части панели. Например, в нашем случае достаточно ввести текст Can, чтобы на панели отобразился единственный компонент Canvas. Можно обойтись и без быстрого поиска, просто выбрав данный компонент в списке All WPF Controls. После выбора компонента Canvas достаточно перетащить его в окно дизайнера. В результате компонент Canvas появится в окне и соответствующий текст будет добавлен в xaml-файл (при этом будет восстановлено представление элемента Window в xaml-файле в виде двух тегов – открывающего <Window> и закрывающего </Window>):

```
<Window x:Class="EVENTS.MainWindow"
    …
    Title="MainWindow" Height="350" Width="525">
  <Canvas HorizontalAlignment="Left" Height="100"
     Margin="150,87,0,0" VerticalAlignment="Top" Width="100"/>
</Window>
```

Здесь и в дальнейшем мы часто будем опускать фрагменты xaml-файла, оставшиеся неизменными, указывая вместо них символ многоточия «…». Измененную часть xaml-файла мы выделили полужирным шрифтом.

Примерный вид окна дизайнера приведен на рис. 2.

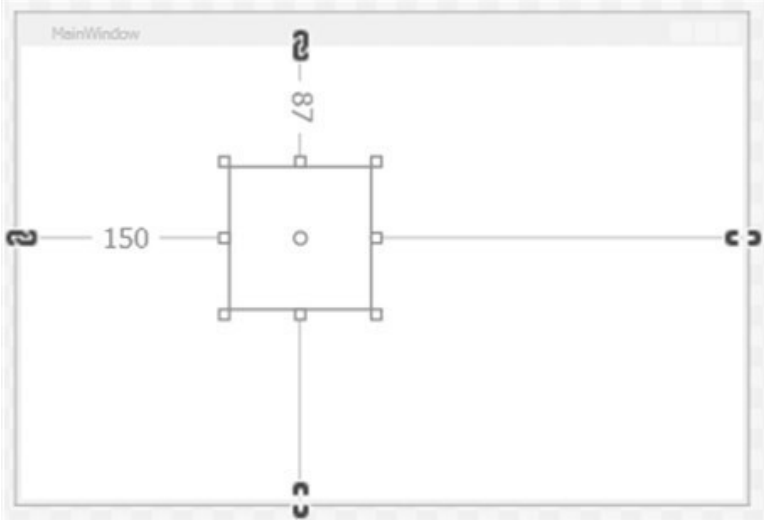

Рис. 2. Окно дизайнера после добавления компонента Canvas

Разумеется, нам не требуется такое размещение компонента Canvas. Необходимо, чтобы он занимал всю клиентскую область окна. Для того чтобы добиться этого, доста-

точно просто *удалить* в xaml-файле все атрибуты элемента Canvas (удаляемые фрагменты будем изображать перечеркнутыми):

```
<Canvas HorizontalAlignment="Left" Height="100"
    Margin="150,87,0,0" VerticalAlignment="Top" Width="100" />
```

Новый вид окна дизайнера приведен на рис. 3.

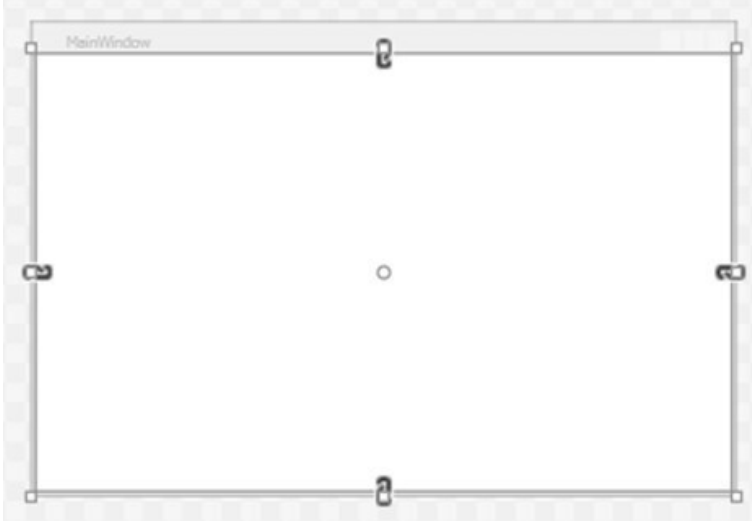

Рис. 3. Окно дизайнера после удаления атрибутов компонента Canvas

Как правило, после добавления в окно какого-либо компонента путем его перетаскивания из панели Toolbox, всегда требуется выполнить действия, связанные с удалением «лишних» атрибутов.

Теперь добавим на компонент Canvas кнопку Button, зацепив ее мышью на панели Toolbox и перетащив в окно. После появления кнопки в окне следует перетащить ее в левый верхний угол окна (при подобном перетаскивании кнопка будет автоматически «притянута» к области, расположенной на расстоянии 10 единиц от левой и верхней границы клиентской области окна, – рис. 4).

Рис. 4. Окно дизайнера после добавления компонента Button

Содержимое xaml-файла изменится следующим образом:

```
<Canvas >
  <Button x:Name="button" Content="Button" Canvas.Left="10"
      Canvas.Top="10" Width="75"/>
</Canvas>
```

Мы видим, что теперь элемент Canvas тоже оформляется в виде парных тегов, так как он содержит дочерний элемент – кнопку.

Обсудим атрибуты, автоматически добавленные к элементу Button. Атрибут с именем x:Name определяет *имя*, с помощью которого можно обращаться к данному компоненту в cs-файле. Это имя будет являться одним из свойств класса MainWindow. Обратите внимание на то, что элемент Canvas аналогичного имени не содержит. Это означает, что в классе MainWindow мы не сможем обращаться по имени к компоненту Canvas. Если это является неудобным, то всегда можно определить имя (или с помощью окна свойств, в котором свойство Name указывается первым, или непосредственно в xaml-файле).

Свойство Content определяет *содержимое* кнопки. В качестве значения свойства Content может указываться не

только строка, но и любой компонент. Более того, на кнопку можно поместить группирующий компонент, в котором, в свою очередь, можно разместить любое количество других компонентов. Это позволяет создавать в WPF-приложении сложные интерфейсные элементы, конструируя их из базовых. Например, можно создать кнопку, содержащую не только текст, но и изображение (в дальнейшем мы воспользуемся этой возможностью – см., например, проект ZOO, п. 7.7).

Следует также обратить внимание на то, что для кнопки не указано свойство Height (хотя свойство Width имеется). Если свойство Height отсутствует, то высота компонента определяется по размерам его содержимого, что в большинстве случаев является оптимальным. Можно было бы удалить и свойство Width, тогда все размеры кнопки будут подстроены под ее содержимое, однако обычно свойство Width указывается, поскольку желательно, чтобы все кнопки в приложении имели одинаковую ширину.

В отличие от компонентов из библиотеки Windows Forms, компоненты библиотеки WPF не имеют свойств Top и Left, определяющих позицию, в которой они размещаются. Это связано с тем, что явное указание позиции компонентов в окне WPF обычно не требуется (положение компонентов определяется другими их свойствами, а также свойствами содержащих их группирующих компонентов). Однако для любого компонента можно задать свойства Top и Left, «полученные» от класса Canvas. Если данный компонент будет

размещен на одном из компонентов типа Canvas, то эти полученные свойства будут учтены при определении его позиции. Возможность подобной «передачи» свойств от одного компонента к другому является одним из аспектов особого механизма, реализованного в WPF и связанного с так называемыми *свойствами зависимости* (dependency properties). Почти все свойства компонентов WPF являются свойствами зависимости, что позволяет их использовать при реализации различных возможностей, доступных в WPF, например, для привязки свойств или определения стилей. Частным случаем свойств зависимости являются *присоединенные свойства* (attached properties), которые, будучи определенными в одном классе, могут использоваться в другом. Свойства Top и Left компонента Canvas – типичный пример присоединенных свойств.

Присоединенные свойства панели Canvas особенно просто указывать в xaml-файле. Однако к ним можно обращаться и в программном коде, что будет продемонстрировано далее.

Итак, в результате добавления новых компонентов в окно наш xaml-файл изменился следующим образом:

```
<Window x:Class="EVENTS.MainWindow"
    …
    Title="MainWindow" Height="350" Width="525">
  <Canvas >
    <Button x:Name="button" Content="Button" Canvas.Left="10"
        Canvas.Top="10" Width="75"/>
  </Canvas>
</Window>
```

Того же результата можно было достичь, просто введя данный текст в xaml-файл, хотя на практике оказывается более удобным добавлять новый компонент с помощью панели Toolbox, а уже затем редактировать связанный с ним текст, добавленный в xaml-файл.

Отредактируем полученный xaml-файл: изменим заголовок окна на текст «Прыгающие кнопки», надпись на кнопке – на «Закрыть», ее имя – на button1 (поскольку в дальнейшем мы добавим к окну еще одну кнопку). Кроме того, укажем для окна свойство WindowStartupLocation, положив его равным CenterScreen (это значение обеспечивает автоматическое центрирование окна программы при ее запуске):

```
<Window x:Class="EVENTS.MainWindow"
    …
    Title="Прыгающие кнопки" Height="350" Width="525"
    WindowStartupLocation="CenterScreen">
  <Canvas >
    <Button x:Name="button1" Content="Закрыть" Canvas.Left="10"
        Canvas.Top="10" Width="75"/>
  </Canvas>
</Window>
```

Хотя свойства можно настраивать с помощью окна Properties, обычно бывает удобнее это делать непосредственно в xaml-файле. Более того, даже добавлять новые свойства в xaml-файл не составляет труда, так как при вводе уже нескольких начальных символов свойства появляется список всех свойств, начинающихся с этих символов, что позволяет быстро завершить ввод имени, нажав клавишу Tab, после чего в xaml-файл будет не только добавлено полное имя свойства, но и вставлены символы =””, а если свойство принимает фиксированный набор значений, то сразу отобразится список этих значений, из которых можно выбрать требуемый (мы могли это заметить, определяя свойство WindowStartupLocation).

В дальнейшем при описании действий, которые требуется выполнить для добавления в окно новых компонентов или изменения их свойств, мы будем просто указывать новое содержимое xaml-файла, выделяя в нем **полужирным шрифтом** новые или измененные фрагменты. Иногда (до-

статочно редко) мы будем также дополнительно помечать фрагменты, которые требуется удалить, оформляя их в виде перечеркнутого текста. Аналогичные способы выделения будем использовать и для фрагментов программного кода на языке C#.

Комментарий

При редактировании xaml-файла оказываются удобными две возможности, связанные с автоматическим добавлением или удалением закрывающих тегов.

(1) Если ввести новый открывающий тег, включая все его атрибуты, то после ввода закрывающей угловой скобки к открывающему тегу будет добавлен закрывающий, а курсор разместится между тегами.

Пример. Предположим, что был введен следующий текст (позиция курсора помечена символом |):

```
<Button Content="Закрыть"|
```

Если теперь ввести символ «>», то текст изменится следующим образом (символ | по-прежнему указывает на позицию курсора):

```
<Button Content="Закрыть">|</Button>
```

(2) Если перед закрывающей угловой скобкой открывающего тега ввести символ «/» (превратив этим действием тег в *комбинированный*), то соответствующий закрывающий тег будет удален из xaml-файла (при этом все дочерние элементы

преобразованного элемента, если они имеются, станут элементами того же уровня, что и преобразованный элемент).

Пример. Если в тексте, полученном в предыдущем примере, перевести курсор на одну позицию влево

```
<Button Content="Закрыть"|></Button>
```

и ввести символ «/», то текст изменится следующим образом:

```
<Button Content="Закрыть"/>|
```

Результат. После запуска программы (для которого достаточно нажать клавишу F5) в центре экрана появится ее окно с кнопкой «Закрыть» (рис. 5).

Рис. 5. Окно приложения EVENTS (первый вариант)

Нажатие на кнопку пока не приводит ни к каким действиям, однако уже сейчас для пользователя доступны все стандартные действия, связанные с управлением окном (сворачиванием, разворачиванием, закрытием, изменением размеров и положения).

Комментарий

При запуске WPF-приложения из среды Visual Studio в режиме Debug поверх окна отображается черная панель с дополнительными средствами отладки (рис. 6).

Рис. 6. Панель с дополнительными отладочными средствами XAML

Поскольку мы не будем использовать эти средства, имеет смысл скрыть панель. Для этого следует выполнить команду меню Tools | Options, в появившемся диалоговом окне Options выбрать раздел Debugging и в этом разделе *снять* флажок Enable UI Debugging Tools for XAML.

1.3. Связывание события с обработчиком

Теперь мы хотим связать определенное действие с нажатием кнопки button1. Для этого можно выполнить следующие шаги:

1) выделите в окне дизайнера кнопку button1;

2) в окне Properties перейдите к разделу со списком событий, нажав на кнопку с изображением молнии:  ;

3) выберите в разделе со списком событий строку Click и выполните на ее пустом поле ввода двойной щелчок мышью;

4) в результате активизируется вкладка редактора с файлом MainWindow.xaml.cs, где появится заготовка для нового метода класса MainWindow – обработчик события Click для компонента button1:

```
private void button1_Click(object sender, RoutedEventArgs e)
{

}
```

5) в эту заготовку надо ввести код, который будет выпол-

няться при нажатии кнопки button1; мы добавим в нее единственный оператор:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    Close();
}
```

Заметим, что соответствующее изменение будет внесено и в xaml-файл:

```
<Button x:Name="button1" Content="Закрыть" Canvas.Left="10"
    Canvas.Top="10" Width="75" Click="button1_Click"/>
```

Именно благодаря заданию атрибута Click в xaml-файле метод button1_Click будет связан с событием Click компонента button1 (при отсутствии такого атрибута метод button1_Click будет считаться обычным методом класса, для выполнения которого требуется его явный вызов).

Описанный выше способ создания нового обработчика события был реализован еще для библиотеки Windows Forms. Однако в WPF-проекте имеется более быстрый способ определения нового обработчика, не требующий использования окна Properties. Необходимо ввести имя события как атрибут соответствующего элемента в xaml-файле (в нашем случае в элемент Button надо ввести текст «Click=»,

причем достаточно набрать несколько начальных символов имени события и воспользоваться для завершения набора выпадающим списком) и после появления рядом с набранным атрибутом выпадающего списка с текстом «New Event Handler» выбрать этот текст (если он еще не выбран) и нажать клавишу Enter. При этом в xaml-файл будет добавлено имя обработчика (в нашем случае button1_Click), а в cs-файле будет создана заготовка для обработчика с этим именем, *хотя перехода к ней не произойдет*, чтобы дать возможность продолжить редактирование xaml-файла. Если в программе уже имеются обработчики, совместимые с тем событием, имя которого введено в xaml-файле, то в выпадающем списке наряду с вариантом «New Event Handler» будут приведены и имена всех таких обработчиков, что позволит быстро связать события для *нескольких* компонентов с *одним* обработчиком (хотя для подобного связывания имеется более удобная возможность, основанная на механизме *маршрутизируемых событий* и описанная в проекте CALC).

Если для какого-либо компонента предполагается определять обработчики, то рекомендуется предварительно задать *имя* для этого компонента, чтобы оно включалось в имена созданных обработчиков.

В дальнейшем вместо детального описания действий по созданию обработчиков событий мы будем просто приводить измененный фрагмент xaml-файла с новыми атрибутами и текст самого обработчика, выделяя добавленные (или

измененные) фрагменты полужирным шрифтом:

```
<Button x:Name="button1" … Click="button1_Click" />
```

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    Close();
}
```

Текст button1_Click мы не только выделяем **полужирным** шрифтом, но и подчеркиваем, чтобы отметить то обстоятельство, что этот текст будет автоматически сгенерирован редактором xaml-файлов после ввода текста Click= и выбора из появившегося списка варианта «New Event Handler» (напомним, что при этом в cs-файле будет создан новый обработчик с указанным именем).

Добавим к нашему проекту еще один обработчик – на этот раз для компонента Canvas (обратите внимание на то, что если обработчик создается для компонента, не имеющего имени, то в имени обработчика по умолчанию используется имя класса этого компонента):

```
<Canvas MouseDown="Canvas_MouseDown" >
```

```
private void Canvas_MouseDown(object sender,
  MouseButtonEventArgs e)
{
    Point p = e.GetPosition(this);
    Canvas.SetLeft(button1, p.X - button1.ActualWidth / 2);
    Canvas.SetTop(button1, p.Y - button1.ActualHeight / 2);
}
```

Кроме того, необходимо задать фон для компонента Canvas:

```
<Canvas MouseDown="Canvas_MouseDown" Background="White" >
```

Результат. Теперь после запуска программы при щелчке на любом месте окна кнопка «Закрыть» услужливо прыгает на указанное место. Нажатие на кнопку «Закрыть» приводит к завершению программы и возврату в среду Visual Studio.

Комментарии

1. Любой компонент WPF реагирует на нажатие мыши только в том случае, если имеет непустой фон. По умолчанию фон некоторых компонентов является пустым (в окне Properties в этом случае свойство Background имеет значение No Brush или вообще не содержит никакого текста). Следует заметить, что, несмотря на вид атрибута Background в xaml-файле, фон определяется не цветом, а особым классом WPF – *кистью* (имеется абстрактный класс Brush, от которого порождается несколько классов-потомков). Указание

цветовой константы означает, что будет использоваться *сплошная* кисть типа SolidBrush с заливкой данного цвета. В последних проектах, описанных в данной книге (TRIGFUNC и HTOWERS), мы познакомимся с *градиентной* кистью, имеющей две разновидности – LinearGradientBrush и RadialGradientBrush.

2. Важной характеристикой любого события, связанного с мышью, является позиция мыши. Для определения этой позиции в обработчиках мыши предусмотрен специальный метод, вызываемый для второго параметра обработчика e: e.GetPosition. Данный метод имеет обязательный параметр, задающий компонент, относительно которого определяется позиция мыши. Мы указали параметр this; это означает, что позиция будет определяться относительно левого верхнего угла клиентской области окна. Заметим, что все размеры в WPF задаются в так называемых *аппаратно-независимых единицах* (одна единица равна 1/96 дюйма) и представляются в виде вещественных чисел типа double (в то время как в библиотеке Windows Forms размеры задавались в экранно-зависимых *пикселах* и представлялись целыми числами).

3. Два последних оператора в обработчике Canvas_MouseDown демонстрируют способ, позволяющий задать в программе *присоединенные свойства* Left и Top, полученные компонентом от его родителя типа Canvas. Обратите внимание на то, что методы SetLeft и SetTop являются статическими и должны вызываться не для конкретного объекта типа

Canvas, а для самого класса. Имеются парные методы Canvas.GetLeft(c) и Canvas.GetTop(c), позволяющие определить текущие значения свойств Left и Top для компонента c, расположенного на компоненте Canvas (эти методы будут использованы во фрагменте программы, добавленном в п. 1.5 при исправлении недочета).

Для задания присоединенных свойств можно также использовать «универсальный» метод SetValue, имеющийся у всех компонентов. Например, два последних оператора в обработчике Canvas_MouseDown можно изменить следующим образом:

```
button1.SetValue(Canvas.LeftProperty,
  p.X - button1.ActualWidth / 2);
button1.SetValue(Canvas.TopProperty,
  p.Y - button1.ActualHeight / 2);
```

Обратите внимание на то, что при указании присоединенного свойства в методе SetValue надо использовать его имя с суффиксом Property (на самом деле это и есть «настоящее» имя статического присоединенного свойства, поскольку подобный суффикс имеют *все* статические свойства зависимости). Интересно отметить, что с помощью метода SetValue с компонентом можно связывать *любые* свойства зависимости, определенные у *любых* типов компонентов (для получения значений этих свойств предназначен метод GetValue, пример использования которого приводится в последнем комментарии к п.

1.5).

4. Для определения *текущих размеров* компонента в программе надо обращаться к свойствам ActualWidth и ActualHeight. Свойства Width и Height для этого использовать нельзя, так как они обычно содержат лишь «рекомендованные» значения размеров, которые учитываются группирующими компонентами при компоновке своих дочерних компонентов (в частности, возможны рекомендованные значения «бесконечность» или NaN). В нашем случае свойства ActualWidth и ActualHeight кнопки используются для того, чтобы отцентрировать кнопку относительно курсора мыши.

1.4. Отсоединение обработчика от события

В начало описания класса MainWindow (перед конструктором public MainWindow()) добавьте новое поле:

```
Random r = new Random();
```

В окно добавьте новую кнопку button2, сделайте ее свойство Content пустой строкой и определите для этой кнопки два обработчика:

```
<Canvas MouseDown="Canvas_MouseDown" Background="White" >
    …
  <Button x:Name="button2" Content="" Canvas.Left="90"
      Canvas.Top="10" Width="75" MouseMove="button2_MouseMove"
      Click="button2_Click" />
</Canvas>
```

```
private void button2_MouseMove(object sender, MouseEventArgs e)
{
```

```
    if (Keyboard.IsKeyDown(Key.LeftCtrl)
      || Keyboard.IsKeyDown(Key.RightCtrl))
        return;
    Point p = e.GetPosition(this);
    Canvas.SetLeft(button2, r.NextDouble() *
      ((Content as Canvas).ActualWidth - 5));
    Canvas.SetTop(button2, r.NextDouble() *
      ((Content as Canvas).ActualHeight - 5));
}
private void button2_Click(object sender, RoutedEventArgs e)
{
    button2.Content = "Изменить";
    button2.MouseMove -= button2_MouseMove;
}
```

Результат. «Дикая» кнопка с пустым заголовком не дает на себя нажать, «убегая» от курсора мыши. Для того чтобы ее «приручить», надо переместить на нее курсор, держа нажатой клавишу Ctrl. После щелчка на дикой кнопке она приручается: на ней появляется заголовок «Изменить», и она перестает убегать от курсора мыши. Следует заметить, что приручить кнопку можно и с помощью клавиатуры, переместив на нее фокус с помощью клавиш со стрелками (или клавиши Tab) и нажав на клавишу пробела.

Недочет. Если попытаться «приручить» кнопку, переместив на нее фокус и нажав клавишу пробела, то перед приручением она прыгает по окну, пока не будет отпущена клавиша пробела. Причины такого поведения непонятны, поскольку нажатие клавиши пробела не должно приводить к

активизации события, связанного с перемещением мыши. Следует, однако, отметить, что нажатие пробела обрабатывается в WPF особым образом, и по этой причине оно может приводить к таким странным эффектам.

Исправление. Дополните условие в методе button2_MouseMove:

```
private void button2_MouseMove(object sender, MouseEventArgs e)
{
    if (Keyboard.IsKeyDown(Key.LeftCtrl)
      || Keyboard.IsKeyDown(Key.RightCtrl)
      || Keyboard.IsKeyDown(Key.Space))
        return;
    Point p = e.GetPosition(this);
    Canvas.SetLeft(button2, r.NextDouble() *
      ((Content as Canvas).ActualWidth - 5));
    Canvas.SetTop(button2, r.NextDouble() *
      ((Content as Canvas).ActualHeight - 5));



}
```

Прирученная кнопка пока ничего не делает. Это будет исправлено в следующем пункте.

Комментарии

1. В данном пункте демонстрируется возможность *отсоединения* метода-обработчика от события, с которым он ранее был связан. Для этого используется операция –=, слева от которой указывается событие, а

справа – тот обработчик, который надо отсоединить от события.

2. В обработчике button2_MouseMove определяются текущие размеры компонента Canvas, чтобы обеспечить случайное перемещение дикой кнопки только в пределах этого компонента (метод r.NextDouble() возвращает случайное вещественное число в полуинтервале [0; 1), при этом вычитание числа 5 гарантирует, что дикая кнопка будет видна на экране хотя бы частично). Заметим, что программа правильно реагирует на изменение размера окна: дикая кнопка всегда перемещается в пределах его текущего размера. Это обеспечивается благодаря тому, что панель Canvas по умолчанию занимает всю клиентскую область своего родителя-окна.

Поскольку мы не присвоили компоненту Canvas имя, нам пришлось обращаться к нему через его родителя, вызвав для окна его свойство Content и, кроме того, выполнив явное приведение типа с помощью операции as. Вместо приведения к типу Canvas можно было бы выполнить приведение к типу FrameworkElement – первому типу в иерархии наследования компонентов, в котором определены свойства, связанные с размерами. Можно было выполнить приведение к типу Panel – непосредственному потомку FrameworkElement, который является предком всех группирующих компонентов. Заметим, что выполнить приведение класса Canvas к типу Control не удастся, так как группирующие компоненты к данному типу не

относятся (потомками Control являются, в частности, компоненты, имеющие свойство Content, например кнопки).

3. Следует обратить внимание на способ, с помощью которого в обработчике button2_MouseMove проверяется, нажата ли клавиша Ctrl. Обычно дополнительная информация о произошедшем событии передается в обработчик с помощью второго параметра e. Например, в обработчике button2_MouseMove с помощью данного параметра (типа MouseEventArgs) можно определить текущую позицию мыши (вызвав метод e.GetPosition) или состояние кнопок мыши (вызвав, например, свойство e.LeftButton и сравнив его с одним из его возможных значений – MouseButtonState.Pressed или MouseButtonState.Released). Однако информацию о нажатых в данный момент *клавишах*

Конец ознакомительного фрагмента.