ПРИЛОЖЕНИЕ A

Программирование с помощью Windows Forms

Со времени первоначального выпуска платформы .NET (в 2001 году) в состав библиотек базовых классов входит API-интерфейс под названием Windows Forms, представленный главным образом сборкой `System.Windows.Forms.dll`. Инструментальный набор Windows Forms предлагает типы, необходимые для построения настольных графических пользовательских интерфейсов, создания специальных элементов управления, манипулирования ресурсами (например, таблицами строк и значками) и решения других задач, связанных с разработкой настольных приложений. Вдобавок отдельный API-интерфейс по имени GDI+ (представленный сборкой `System.Drawing.dll`) предоставляет дополнительные типы, которые позволяют программистам генерировать двухмерную графику, взаимодействовать с сетевыми принтерами и манипулировать графическими данными.

API-интерфейсы Windows Forms (и GDI+) остались в .NET 4.0 и последующих версиях и будут находиться в библиотеках базовых классов еще некоторое время (а может быть и всегда). Однако в выпуске .NET 3.0 появился совершенно новый инструментальный набор для построения графических пользовательских интерфейсов под названием Windows Presentation Foundation (WPF). Как было показано в книге, WPF предлагает мощные средства, которые позволяют строить передовые пользовательские интерфейсы, и поэтому он стал предпочитаемым API-интерфейсом для создания современных графических пользовательских интерфейсов .NET.

Тем не менее, целью настоящего приложения является ознакомительный тур по традиционному API-интерфейсу Windows Forms. Одна из причин связана с тем, что это полезно для понимания исходной модели программирования: существует довольно много приложений Windows Forms, которые придется сопровождать в течение определенного времени. Кроме того, многие графические пользовательские интерфейсы попросту не нуждаются во всей той мощи, что предлагает WPF. Когда необходимо создавать более традиционные пользовательские интерфейсы, которым не требуются разнообразные излишества, часто вполне достаточно API-интерфейса Windows Forms.

В этом приложении вы изучите модель программирования Windows Forms, поработаете с интегрированными визуальными конструкторами Visual Studio, опро-

буете многочисленные элементы управления Windows Forms и ознакомитесь с программированием графики посредством GDI+. Вы также соберете все это в единое целое, упаковав в несложное приложение рисования.

На заметку! Вот одно из доказательств того, что Windows Forms не исчезнет в ближайшее время: версия .NET 4.0 поставлялась с совершенно новой сборкой Windows Forms, `System.Windows.Forms.DataVisualization.dll`. Эту библиотеку можно использовать для внедрения в программы функциональности построения диаграмм, дополняемых аннотациями, трехмерной графики и поддержкой проверки попадания. Здесь новый API-интерфейс Windows Forms из .NET 4.0 не рассматривается, но если вас интересует информация о нем, просмотрите документацию по пространству имен `System.Windows.Forms.DataVisualization.Charting`.

Пространства имен Windows Forms

API-интерфейс Windows Forms состоит из сотен типов (классов, интерфейсов, структур, перечислений и делегатов), большинство из которых организованы в разнообразные пространства имен внутри сборки `System.Windows.Forms.dll`. На рис. А.1 показаны эти пространства имен, отображаемые в браузере объектов Visual Studio.

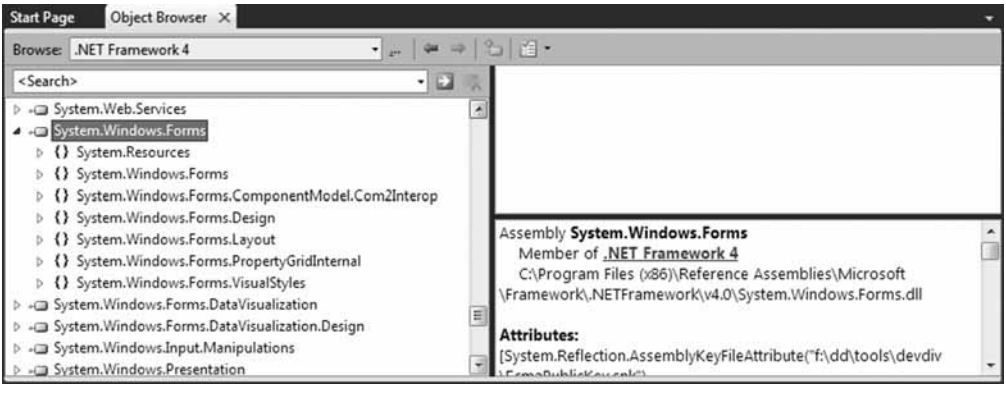

Рис. А.1. Пространства имен сборки `System.Windows.Forms.dll`

Несомненно, самым важным пространством имен Windows Forms является `System.Windows.Forms`. С высокоуровневой точки зрения типы в этом пространстве имен можно разбить на следующие крупные категории.

- *Основная инфраструктура.* Типы, представляющие основные операции программ Windows Forms (`Form` и `Application`), а также различные типы, которые предназначены для взаимодействия с унаследованными элементами ActiveX и новыми специальными элементами управления WPF.
- *Элементы управления.* Типы, которые применяются для создания графических пользовательских интерфейсов (например, `Button`, `MenuStrip`, `ProgressBar` и `DataGridView`) и являются производными от базового класса `Control`. Элементы управления допускают конфигурирование на этапе проектирования и видимы (по умолчанию) во время выполнения.

- *Компоненты.* Типы, которые не являются производными от базового класса `Control`, но все равно могут предоставлять программам Windows Forms визуальные возможности (например, `ToolTip` и `ErrorProvider`). Многие компоненты (скажем, `Timer` и `System.ComponentModel.BackgroundWorker`) не будут видимыми во время выполнения, но могут конфигурироваться на этапе проектирования.
- *Общие диалоговые окна.* В Windows Forms есть несколько готовых диалоговых окон для выполнения распространенных операций (например, `OpenFileDialog`, `PrintDialog` и `ColorDialog`). Вполне ожидаемо, можно строить собственные диалоговые окна, если стандартные диалоговые окна не удовлетворяют существующим потребностям.

Учитывая, что суммарное число типов внутри пространства имен `System.Windows.Forms` значительно превышает сотню, было бы излишней тратой времени (и бумаги) перечислять абсолютно все члены семейства Windows Forms. Однако по мере чтения данного приложения вы сформируете устойчивый фундамент, чтобы двигаться дальше самостоятельно. В любом случае дополнительную информацию можно найти в документации .NET Framework SDK.

Построение простого приложения Windows Forms

Как и можно было ожидать, современные IDE-среды .NET (такие как Visual Studio или Visual Studio Express) содержат многочисленные визуальные конструкторы форм, визуальные редакторы и интегрированные инструменты генерации кода (мастера), которые облегчают создание приложений Windows Forms. Такие инструменты исключительно удобны, но они препятствуют процессу изучения Windows Forms, потому что генерируют большой объем стереотипного кода, который может затенять основную объектную модель. С учетом этого первый пример приложения Windows Forms будет проектом консольного приложения.

Начните с создания проекта консольного приложения по имени `SimpleWinFormsApp`. Затем выберите пункт меню Project⇨Add Reference (Проект⇨Добавить ссылку) и на вкладке .NET открывшегося диалогового окна установите ссылки на сборки `System.Windows.Forms.dll` и `System.Drawing.dll`. Далее поместите в файл `Program.cs` следующий код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// Пространства имен, минимально необходимые для Windows Forms.
using System.Windows.Forms;
namespace SimpleWinFormsApp
{
  // Это объект приложения.
  class Program
  {
    static void Main(string[] args)
    {
      Application.Run(new MainWindow());
    }
  }
  // Это главное окно.
  class MainWindow : Form {}
}
```

На заметку! Когда IDE-среда Visual Studio обнаруживает класс, который расширяет `System.Windows.Forms.Form`, она пытается открыть связанный визуальный конструктор с графическим пользовательским интерфейсом (при условии, что это первый класс в файле кода C#). Двойной щелчок на файле `Program.cs` в окне Solution Explorer приводит к открытию визуального конструктора, но в настоящий момент не делайте его! Вы будете работать с визуальным конструктором Windows Forms в следующем примере, а пока щелкните правой кнопкой мыши на файле с кодом C# в окне Solution Explorer и выберите в контекстном меню пункт View Code (Просмотреть код).

Этот код представляет простейшее приложение Windows Forms, какое только можно построить. В качестве абсолютного минимума необходим класс, расширяющий базовый класс `Form`, и метод `Main()` для вызова статического метода `Application.Run()` (классы `Form` и `Application` будут более подробно обсуждаться позже в приложении). Запустив приложение прямо сейчас, вы обнаружите, что располагаете окном, поддерживающим сворачивание, разворачивание, закрытие и изменение размеров, которое находится поверх имеющихся окон (рис. A.2).

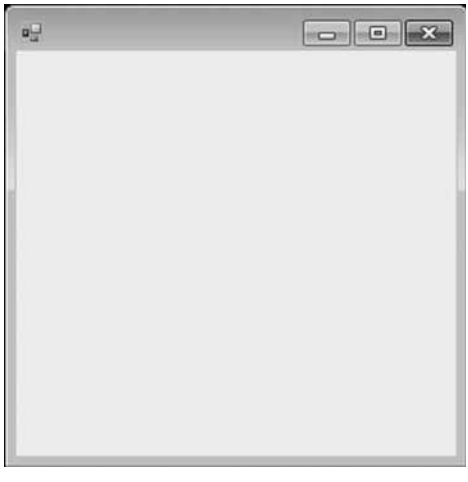

Рис. A.2. Простое приложение Windows Forms

На заметку! В процессе запуска программы вы заметите под верхним окном очертания окна командной строки. Причина в том, что в случае создания консольного приложения компилятору C# передается флаг `/target:exe`. Чтобы предотвратить отображение окна командной строки, измените флаг на `/target:winexe`, дважды щелкнув на значке Properties (Свойства) в окне Solution Explorer и на вкладке Application (Приложение) открывшегося окна свойств выберите в списке Output Type (Тип вывода) вариант Windows Application (Приложение Windows).

Приложение в его текущем виде не особенно впечатляет, но оно демонстрирует, насколько простым может быть приложение Windows Forms. Чтобы слегка его улучшить, можете добавить в класс `MainWindow` специальный конструктор, который позволяет вызывающему коду устанавливать разнообразные свойства отображаемого окна:

```
// Это главное окно.
class MainWindow : Form
{
  public MainWindow() {}
  public MainWindow(string title, int height, int width)
  {
    // Установить различные свойства родительского класса.
    Text = title;
    Width = width;
    Height = height;
    // Унаследованный метод для вывода окна в центре экрана.
    CenterToScreen();
  }
}
```

Теперь можно модифицировать вызов `Application.Run()`:

```
static void Main(string[] args)
{
  Application.Run(new MainWindow("My Window", 200, 300));
}
```

Это шаг в правильном направлении, но любое полезное окно должно содержать различные элементы пользовательского интерфейса (например, меню, строки состояния и кнопки), делая возможным ввод со стороны пользователей. Чтобы понять, как производный от `Form` тип может содержать такие элементы, необходимо разобраться в роли свойства `Controls` и лежащей в основе коллекции элементов управления.

Заполнение коллекции элементов управления

Базовый класс `System.Windows.Forms.Control` (находящийся в цепочке наследования типа `Form`) определяет свойство по имени `Controls`. Оно предоставляет доступ к специальной коллекции `ControlCollection`, вложенной в класс `Control`. Данная коллекция содержит ссылки на все элементы пользовательского интерфейса, поддерживаемые этим производным типом. Подобно другим контейнерам тип `ControlCollection` предлагает несколько методов для вставки, удаления и поиска конкретного виджета пользовательского интерфейса (табл. А.1).

Таблица А.1. Члены `ControlCollection`

Член	Описание
`Add()` `AddRange()`	Эти члены вставляют в коллекцию новый объект производного от `Control` типа (или массив таких объектов)
`Clear()`	Этот член удаляет все элементы из коллекции
`Count`	Этот член возвращает количество элементов в коллекции
`Remove()` `RemoveAt()`	Эти члены удаляют элемент управления из коллекции

Когда нужно заполнить пользовательский интерфейс производного от `Form` типа, обычно выполняется одна и та же последовательность шагов:

- определить переменную-член заданного элемента пользовательского интерфейса внутри производного от `Form` класса;
- сконфигурировать внешний вид и поведение элемента пользовательского интерфейса;
- добавить элемент пользовательского интерфейса в контейнер `ControlCollection` формы, используя вызов `Controls.Add()`.

Предположим, что необходимо обновить класс `MainWindow` для поддержки системы меню File⇨Exit (Файл⇨Выход). Ниже показаны изменения, которые понадобится внести:

```
class MainWindow : Form
{
  // Члены для простой системы меню.
  private MenuStrip mnuMainMenu = new MenuStrip();
  private ToolStripMenuItem mnuFile = new ToolStripMenuItem();
  private ToolStripMenuItem mnuFileExit = new ToolStripMenuItem();

  public MainWindow(string title, int height, int width)
  {
    ...
    // Метод для создания системы меню.
    BuildMenuSystem();
  }

  private void BuildMenuSystem()
  {
    // Добавить в главное меню пункт File.
    mnuFile.Text = "&File";
    mnuMainMenu.Items.Add(mnuFile);

    // Добавить в меню File пункт Exit.
    mnuFileExit.Text = "E&xit";
    mnuFile.DropDownItems.Add(mnuFileExit);
    mnuFileExit.Click += (o, s) => Application.Exit();

    // Установить меню для этой формы.
    Controls.Add(this.mnuMainMenu);
    MainMenuStrip = this.mnuMainMenu;
  }
}
```

Обратите внимание, что теперь тип `MainWindow` содержит три новых переменных-члена. Тип `MenuStrip` представляет всю систему меню, а отдельный объект `ToolStripMenuItem` — любые пункты меню верхнего уровня (скажем, File) либо элементы подменю (например, Exit), поддерживаемые окном.

Система меню конфигурируется внутри вспомогательного метода `BuildMenuSystem()`. Текст каждого объекта `ToolStripMenuItem` управляется свойством `Text`; каждому пункту меню назначается строковый литерал, который содержит символ амперсанда. Как вам может быть известно, такой синтаксис устанавливает клавиатурное сокращение с клавишей <Alt>. Так, нажатие <Alt+F>

активизирует меню File, а <Alt+X> — пункт меню Exit. Кроме того, обратите внимание, что объект `ToolStripMenuItem` (`mnuFile`) меню File добавляет подпункты с применением свойства `DropDownItems`. Сам объект `MenuStrip` добавляет пункты меню верхнего уровня, используя свойство `Items`.

Построенную систему меню можно добавить в коллекцию элементов управления (через свойство `Controls`). Затем объект `MenuStrip` присваивается свойству `MainMenuStrip` формы. Этот шаг может показаться избыточным, но наличие специфичного свойства, такого как `MainMenuStrip`, дает возможность динамически выбирать, какую систему меню показывать пользователю. Отображаемое меню можно изменять на основе пользовательских предпочтений или настроек безопасности.

Еще один интересный аспект связан с обработкой события `Click` пункта меню File⇨Exit; это помогает перехватить момент выбора пользователем данного пункта. Событие `Click` работает в сочетании со стандартным типом делегата по имени `System.EventHandler`. Данное событие может вызывать только методы, которые принимают `System.Object` в первом параметре и `System.EventArgs` — во втором. Здесь применяется лямбда-выражение для завершения всего приложения посредством статического метода `Application.Exit()`.

Перекомпилировав и запустив приложение, вы обнаружите, что простое окно получило специаальную систему меню (рис. А.3).

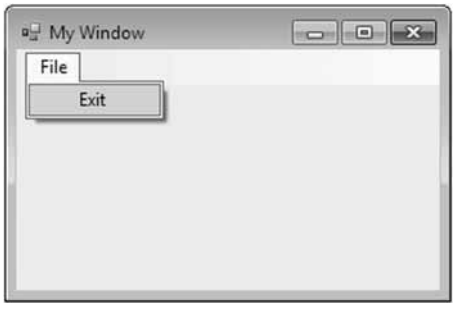

Рис. А.3. Простое окно с простой системой меню

Роль типов `System.EventArgs` и `System.EventHandler`

Тип `System.EventHandler` — один из многих делегатов, используемых в API-интерфейсах Windows Forms (и ASP.NET) во время процесса обработки событий. Как вы уже видели, этот делегат может указывать только на методы, где первый аргумент имеет тип `System.Object`, а второй является ссылкой на объект, отправивший событие. Предположим, что нужно изменить реализацию лямбда-выражения следующим образом:

```
mnuFileExit.Click += (o, s) =>
  {
    MessageBox.Show(string.Format("{0} sent this event", o.ToString()));
    Application.Exit();
  };
```

Отображение в окне сообщения показанной ниже строки подтверждает то, что элемент mnuFileExit отправил событие:

```
"E&xit {0} sent this event"
```

Вас может интересовать, какой цели служит второй аргумент, System.EventArgs. В действительности он мало что привносит, т.к. просто расширяет тип Object и практически не добавляет дополнительной функциональности:

```
public class EventArgs
{
  public static readonly EventArgs Empty;
  static EventArgs();
  public EventArgs();
}
```

Тем не менее, этот тип полезен в общей схеме обработки событий в .NET, т.к. он является родительским для многих производных типов. Например, тип MouseEventArgs расширяет EventArgs для предоставления деталей, касающихся текущего состояния мыши. Тип KeyEventArgs также расширяет EventArgs, чтобы предоставить детали относительно состояния клавиатуры (такие как нажатая клавиша), тип PaintEventArgs расширяет EventArgs для выдачи данных, связанных с графикой, и т.д. Вы столкнетесь с многочисленными потомками EventArgs (и с делегатами, которых их применяют) не только при работе с Windows Forms, но и во время использования API-интерфейсов WPF и ASP.NET.

Хотя вы можете продолжить наращивание функциональности в классе MainWindow (скажем, добавить строки состояния и диалоговые окна) с применением простого текстового редактора, это может быстро утомить, поскольку придется вручную писать стереотипную логику настройки элементов управления. К счастью, IDE-среда Visual Studio предлагает множество визуальных конструкторов, которые позаботятся о таких деталях. При использовании этих средств в оставшихся примерах приложения не забывайте, что они просто генерируют обыденный код C#; в них нет ничего магического.

Исходный код. Проект SimpleWinFormsApp находится в подкаталоге Appendix_A.

Шаблон проекта приложения Windows Forms в Visual Studio

Применение инструментов визуального проектирования Windows Forms в Visual Studio обычно начинается с выбора шаблона проекта Windows Forms Application (Приложение Windows Forms), используя пункт меню File⇨New Project (Файл⇨Новый проект). Чтобы освоится с основными инструментами визуального проектирования Windows Forms, создайте новое приложение по имени SimpleVSWinFormsApp (рис. A.4).

Поверхность визуального конструктора

Прежде чем заняться построением более интересных приложений Windows, имеет смысл воссоздать предыдущий пример, задействовав инструменты визу-

ального проектирования. После создания нового проекта приложения Windows Forms вы заметите, что IDE-среда Visual Studio отобразит поверхность визуального конструктора, на которую можно перетаскивать любое количество элементов управления.

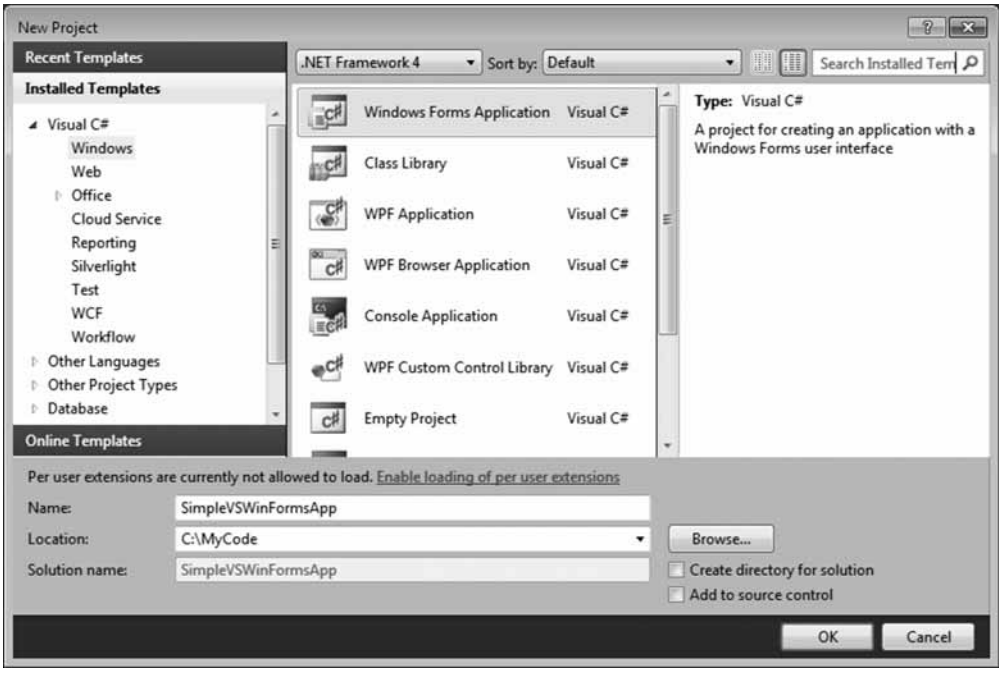

Рис. А.4. Шаблон проекта Windows Forms Application в Visual Studio

Тот же конструктор можно применять для настройки начальных размеров окна, просто изменяя размеры самой формы с помощью манипуляторов захвата (рис. А.5).

Рис. А.5. Визуальный конструктор форм

Если вы захотите сконфигурировать внешний вид окна (а также любого элемента, помещенного на поверхность визуального конструктора форм), то можете делать это в окне Properties (Свойства). Подобно проекту приложения Windows Presentation Foundation, окно Properties используется для присваивания значений свойствам, а также для установки обработчиков событий для элемента, выделенного в текущий момент на поверхности конструктора (конфигурируемый элемент выбирается из раскрывающегося списка в верхней части окна Properties).

Пока форма не содержит ничего, поэтому список содержит только первоначальный элемент `Form` со стандартным именем `Form1`, как видно в допускающем только чтение свойстве `Name` (рис. А.6).

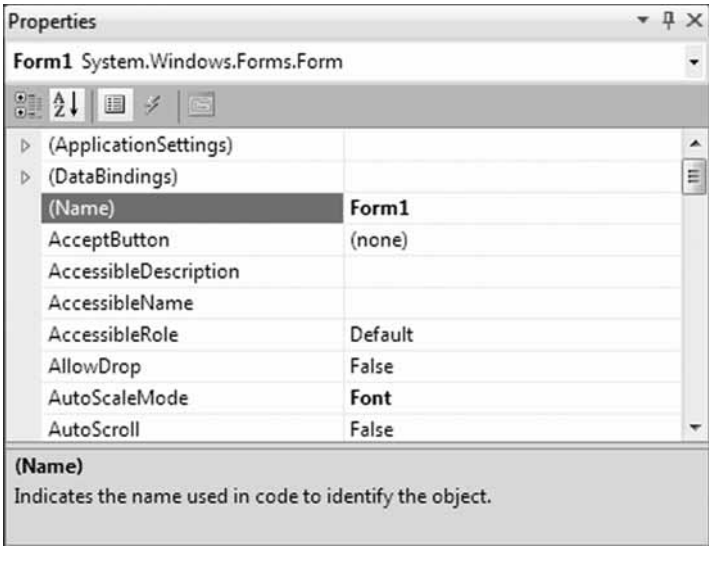

Рис. А.6. Окно Properties, предназначенное для установки свойств и обработчиков событий

На заметку! Окно Properties можно сконфигурировать на отображение содержимого по категориям или в алфавитном порядке с применением первых двух кнопок, расположенных прямо под раскрывающимся списком. Рекомендуется выбрать упорядочение по алфавиту, потому что так быстрее находить необходимое свойство или событие.

Следующим важным элементом визуального конструктора является окно Solution Explorer (Проводник решений). Все проекты Visual Studio поддерживают это окно, но оно особенно удобно при построении приложений Windows Forms для быстрого изменения имени файла и связанного класса любого окна и просмотра файла, который содержит сопровождаемый конструктором код (вы узнаете о нем немного позже). Пока что щелкните правой кнопкой мыши на значке `Form1.cs` и выберите в контекстном меню пункт Rename (Переименовать). Назовите начальное окно более осмысленно — `MainWindow.cs`. Среда IDE запросит о необходимости изменения имени начального класса; ответьте утвердительно.

Разбор первоначальной формы

Перед построением системы меню необходимо выяснить, что конкретно IDE-среда Visual Studio создала по умолчанию. Щелкните правой кнопкой мыши на значке `MainWindow.cs` в окне Solution Explorer и выберите в контекстном меню пункт View Code (Просмотреть код). Обратите внимание, что форма определена как частичный тип, что позволяет единственному типу быть определенным в нескольких файлах кода. Кроме того, конструктор формы вызывает метод `InitializeComponent()`, а сгенерированный объект "является" `Form`:

```
namespace SimpleVSWinFormsApp
{
  public partial class MainWindow : Form
  {
    public MainWindow()
    {
      InitializeComponent();
    }
  }
}
```

Как и можно было ожидать, метод `InitializeComponent()` определен в отдельном файле, который завершает определение частичного класса. По соглашению имя такого файла всегда заканчивается на `.Designer.cs`, а перед этим находится имя связанного файла кода C#, содержащего производный от `Form` тип. Используя окно Solution Explorer, откройте файл `MainWindow.Designer.cs`. Взгляните на следующий код (для простоты из него убраны комментарии; в зависимости от настроек, выполненных в окне Properties, код может слегка отличаться):

```
partial class MainWindow
{
  private System.ComponentModel.IContainer components = null;
  protected override void Dispose(bool disposing)
  {
    if (disposing && (components != null))
    {
      components.Dispose();
    }
    base.Dispose(disposing);
  }
  private void InitializeComponent()
  {
    this.SuspendLayout();
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(422, 114);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
  }
}
```

Переменная-член типа IContainer и метод Dispose() — это всего лишь инфраструктура, применяемая инструментами визуального проектирования Visual Studio. Однако обратите внимание, что метод InitializeComponent() используется не только конструктором формы во время выполнения; IDE-среда Visual Studio вызывает его во время проектирования для корректной визуализации пользовательского интерфейса на поверхности визуального конструктора форм. Чтобы удостовериться в этом, модифицируйте значение свойства Text текущей формы, указав "My Main Window". После активизации визуального конструктора заголовок формы соответствующим образом изменится.

В случае применения инструментов визуального проектирования (окна Properties или визуального конструктора форм) IDE-среда автоматически обновляет код метода InitializeComponent(). Для демонстрации этого аспекта инструментов визуального проектирования Windows Forms сделайте активным окно визуального конструктора форм и найдите в окне Properties свойство Opacity. Измените его значение на 0.8 (т.е. 80%); после компиляции и запуска программы окно станет слегка прозрачным. А теперь еще раз просмотрите реализацию метода InitializeComponent():

```
private void InitializeComponent()
{
  ...
  this.Opacity = 0.8;
}
```

При любой практической работе вы должны игнорировать файлы *.Designer.cs и позволить IDE-среде автоматически поддерживать их при построении приложения Windows Forms с использованием Visual Studio. Если вы вставите в InitializeComponent() синтаксически (или логически) некорректный код, то можете нарушить работу визуального конструктора. Кроме того, Visual Studio часто переформатирует этот метод на этапе проектирования. Таким образом, если вы добавите в метод InitializeComponent() специальный код, то IDE-среда может просто удалить его. Не забывайте, что каждое окно приложения Windows Forms составлено с применением частичных классов.

Разбор класса Program

Помимо кода реализации для первоначального типа, производного от Form, в типах проекта Windows Forms Application также содержится статический класс по имени Program, который определяет точку входа программы, Main():

```
static class Program
{
  [STAThread]
  static void Main()
  {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainWindow());
  }
}
```

Метод `Main()` вызывает `Application.Run()` и несколько других методов типа `Application`, чтобы установить ряд базовых параметров визуализации. Обратите внимание, что метод `Main()` снабжен атрибутом `[STAThread]`, который сообщает исполняющей среде, что если данный поток создаст за время своей жизни какие-либо классические объекты COM (включая унаследованные элементы управления ActiveX), то их следует поместить в специальную область, обслуживаемую COM, под названием *однопоточный апартамент*. В сущности, это гарантирует, что объекты COM будут безопасными к потокам, даже если автор объекта COM не предусмотрел для такого случая специальный код.

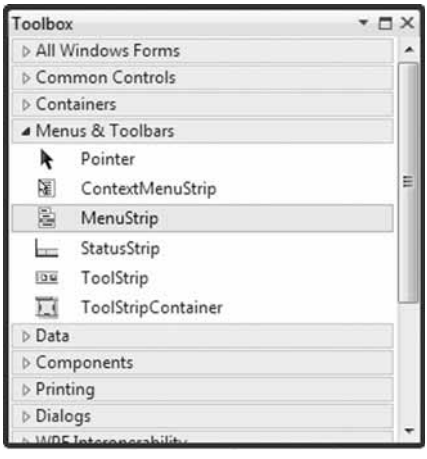

Рис. А.7. Элементы управления Windows Forms, которые можно добавлять на поверхность визуального конструктора

Визуальное построение системы меню

Чтобы завершить обзор инструментов визуального проектирования Windows Forms и перейти к более иллюстративным примерам, активизируйте окно визуального конструктора форм, отыщите окно Toolbox (Панель инструментов) Visual Studio и внутри узла Menus & Toolbars (Меню и панели инструментов) найдите элемент MenuStrip (рис. А.7).

Перетащите элемент управления `MenuStrip` в верхнюю часть визуального конструктора форм. Среда Visual Studio отреагирует открытием редактора меню. Внимательно присмотревшись к этому редактору, в верхнем правом углу элемента управления можно заметить значок с небольшим треугольником. Щелчок на нем открывает контекстно-чувствительный встроенный редактор, который позволяет устанавливать значения сразу нескольких свойств (похожие редакторы имеются у многих элементов управления Windows Forms). К примеру, щелкните на ссылке Insert Standard Items (Вставить стандартные элементы), как показано на рис. А.8.

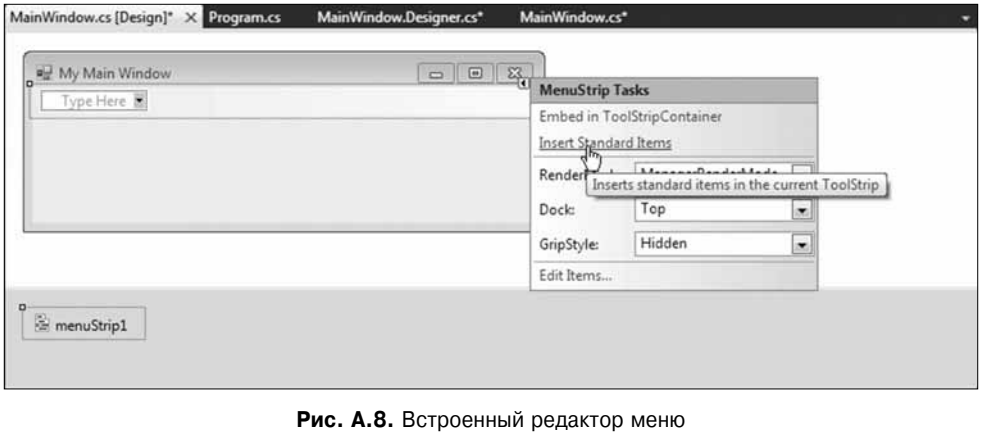

Рис. А.8. Встроенный редактор меню

В этом примере IDE-среда Visual Studio создала для вас полную систему меню. Теперь откройте сопровождаемый конструктором файл (`MainWindow.Designer.cs`) и обратите внимание, что в метод `InitializeComponent()` были добавлены многочисленные строки кода, а также несколько новых переменных-членов, которые представляют систему меню. И, наконец, возвратитесь в окно визуального конструктора и отмените последнюю операцию, нажав комбинацию <Ctrl+Z>. Вы опять попадете в редактор меню, а сгенерированный код будет удален. С использованием визуального конструктора меню введите самый верхний пункт меню File (Файл) и его подпункт Exit (Выход), как показано на рис. A.9.

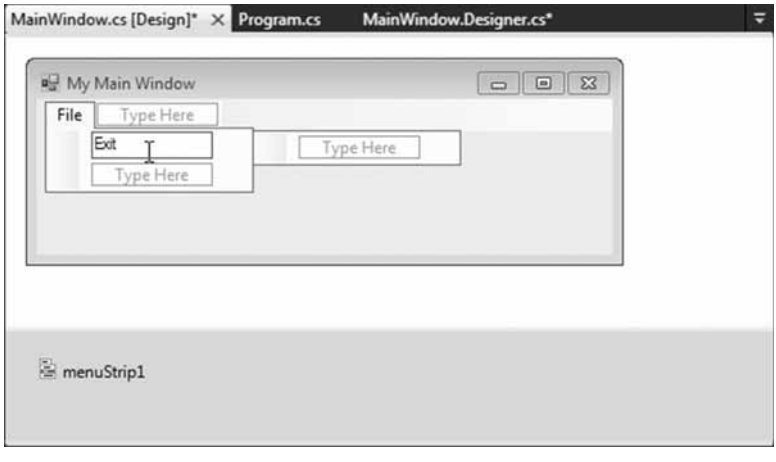

Рис. A.9. Ручное построение системы меню

Заглянув в метод `InitializeComponent()`, вы обнаружите код того же вида, который вы вводили вручную в первом примере приложения. В заключение упражнения возвратитесь в окно визуального конструктора форм и щелкните на кнопке со значком молнии в окне Properties. Отобразятся все события, которые можно обработать для выбранного элемента. Удостоверьтесь, что выбран элемент меню Exit (со стандартным именем `exitToolStripMenuItem`), и найдите событие `Click` (рис. A.10).

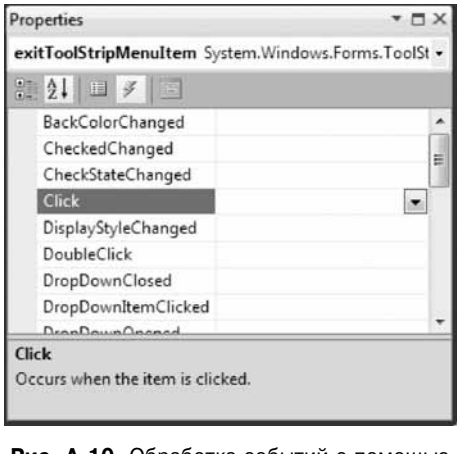

Рис. A.10. Обработка событий с помощью IDE-среды

В этот момент можно ввести имя метода, подлежащего вызову, когда на элементе осуществляется щелчок. Или, если вы считаете ввод утомительным, то просто дважды щелкните на перечисленном событии в окне Properties, что позволит IDE-среде выбрать имя обработчика события (в виде *`ИмяЭлемента_ИмяСобытия`*`()`). В любом случае IDE-среда создает код заглушки, который можно заполнить конкретной реализацией:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    InitializeComponent();
    CenterToScreen();
  }
  private void exitToolStripMenuItem_Click(object sender, EventArgs e)
  {
    Application.Exit();
  }
}
```

При желании просмотрите код `InitializeComponent()`, где также учтены все последние изменения:

```
this.exitToolStripMenuItem.Click +=
  new System.EventHandler(this.exitToolStripMenuItem_Click);
```

К настоящему времени вы должны уже освоиться с IDE-средой при построении приложений Windows Forms. Наряду с тем, что есть много дополнительных клавиатурных сокращений, редакторов и интегрированных мастеров генерации кода, предоставленных сведений вполне достаточно для самостоятельного продолжения работы.

Внутреннее устройство формы

Итак, вы узнали, как создавать простые приложения Windows Forms с помощью Visual Studio (или без нее). Теперь пора приступать к подробным исследованиям типа `Form`. В мире Windows Forms тип `Form` представляет любое окно в приложении, включая главное окно самого верхнего уровня, дочерние окна приложений с многодокументным интерфейсом (multiple document interface — MDI), а также модальные и немодальные диалоговые окна. Как показано на рис. А.11, тип `Form` наследует много функциональности от своих родительских классов и реализует многочисленные интерфейсы.

В табл. А.2 приведен высокоуровневый обзор родительских классов в цепочке наследования `Form`.

Хотя полное происхождение типа `Form` охватывает многочисленные базовые классы и интерфейсы, имейте в виду, что вы вовсе *не* обязаны изучать роль абсолютно каждого члена всех родительских классов или реализованных интерфейсов, чтобы стать профессиональным разработчиком приложений Windows Forms. В действительности большинство членов (в частности, свойств и событий) вы можете легко устанавливать с помощью окна Properties в Visual Studio. Это значит, что важно понимать функциональность, предоставляемую родительскими классами `Control` и `Form`.

Таблица A.2. Базовые классы в цепочке наследования `Form`

Родительский класс	Описание
`System.Object`	Подобно любому классу в .NET класс `Form` является `object`
`System.MarshalByRefObject`	К типам, производным от этого класса, можно обращаться удаленно с помощью *ссылки* на удаленный тип (а не локальной копии)
`System.ComponentModel.Component`	Предоставляет стандартную реализацию интерфейса `IComponent`. В мире .NET компонент — это тип, поддерживающий редактирование на этапе проектирования, но он не обязательно будет видимым во время выполнения
`System.Windows.Forms.Control`	Определяет общие члены пользовательского интерфейса для всех элементов управления Windows Forms, включая сам тип `Form`
`System.Windows.Forms.ScrollableControl`	Определяет поддержку горизонтальных и вертикальных полос прокрутки, а также членов, которые позволяет управлять окном просмотра, отображаемым в прокручиваемой области
`System.Windows.Forms.ContainerControl`	Предоставляет функциональность управления фокусом ввода элементам, которые могут служить контейнерами для других элементов управления
`System.Windows.Forms.Form`	Представляет любую специальную форму, дочернюю форму MDI или диалоговое окно

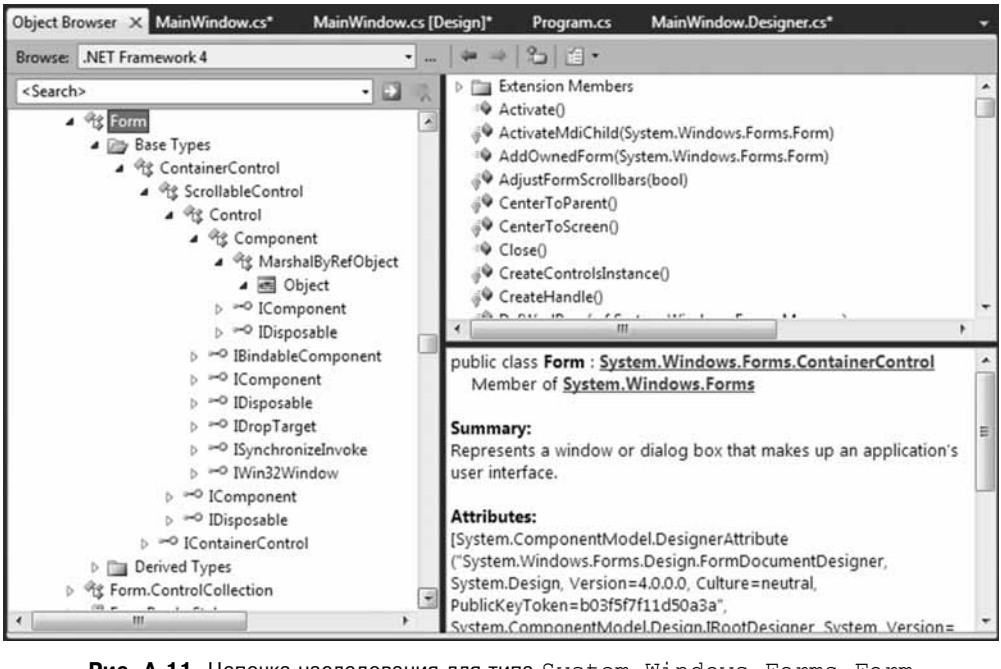

Рис. A.11. Цепочка наследования для типа `System.Windows.Forms.Form`

Функциональность класса Control

Класс System.Windows.Forms.Control устанавливает общие линии поведения, требуемые любым типом графического пользовательского интерфейса. Основные члены класса Control позволяют настраивать размер и позицию элемента, захватывать ввод с клавиатуры и мыши, получать или задавать фокус/видимость члена и т.д. В табл. А.3 приведены наиболее интересные свойства, которые сгруппированы по связанной функциональности.

Таблица А.3. Основные свойства типа Control

Свойство	Описание
BackColor ForeColor BackgroundImage Font Cursor	Эти свойства определяют основные характеристики элемента управления (цвет, шрифт для текста и вид курсора мыши, когда он находится над элементом)
Anchor Dock AutoSize	Эти свойства задают положение элемента управления внутри контейнера
Top Left Bottom Right Bounds ClientRectangle Height Width	Эти свойства указывают текущие размеры элемента управления
Enabled Focused Visible	Эти свойства инкапсулируют булевское значение, которое указывает состояние текущего элемента управления
ModifierKeys	Это статическое свойство проверяет текущее состояние клавиш-модификаторов (<Shift>, <Ctrl> и <Alt>) и возвращает данное состояние как тип Keys
MouseButtons	Это статическое свойство проверяет текущее состояние кнопок мыши (левой, правой и средней) и возвращает данное состояние как тип MouseButtons
TabIndex TabStop	Эти свойства позволяют сконфигурировать очередность обхода по нажатию клавиши <Tab>
Opacity	Это свойство определяет прозрачность элемента управления (0.0 — полностью прозрачный, 1.0 — полностью непрозрачный)
Text	Это свойство содержит строковые данные, ассоциированные с элементом управления
Controls	Это свойство предоставляет доступ к строго типизированной коллекции (например, ControlsCollection), которая содержит все дочерние элементы управления текущего элемента управления

Как и можно было догадаться, в классе Control также определено несколько событий, которые помимо прочего позволяют перехватывать действия, связанные с мышью, клавиатурой, отображением и перетаскиванием. В табл. А.4 кратко описаны полезные события, сгруппированные по связанной функциональности.

Таблица A.4. События типа `Control`

Событие	Описание

`Click`
`DoubleClick`
`MouseEnter`
`MouseLeave`
`MouseDown`
`MouseUp`
`MouseMove`
`MouseHover`
`MouseWheel`
	Эти события позволяют взаимодействовать с мышью

`KeyPress`
`KeyUp`
`KeyDown`
	Эти события позволяют взаимодействовать с клавиатурой

`DragDrop`
`DragEnter`
`DragLeave`
`DragOver`
	Эти события позволяют отслеживать действие перетаскивания

`Paint`	Это событие позволяет взаимодействовать со службами графической визуализации GDI+

Наконец, в базовом классе `Control` определено несколько методов, которые позволяют взаимодействовать с любым типом, производным от `Control`. При исследовании методов типа `Control` обратите внимание, что имена многих из них имеют префикс `On`, за которым следует имя специфического события (например, `OnMouseMove`, `OnKeyUp` и `OnPaint`). Каждый такой виртуальный метод с префиксом `On` является стандартным обработчиком для соответствующего события. Если вы переопределите любой из этих виртуальных методов, то получите возможность выполнять всю необходимую пред- или постобработку события перед или после вызова стандартной реализации родительского класса:

```
public partial class MainWindow : Form
{
  protected override void OnMouseDown(MouseEventArgs e)
  {
    // Предусмотреть здесь специальный код обработки события MouseDown.
    // По завершении вызвать родительскую реализацию.
    base.OnMouseDown(e);
  }
}
```

В определенных обстоятельствах это может быть удобно (особенно, если вы хотите построить специальный элемент управления, производный от стандартного элемента), но события будут часто обрабатываться с применением стандартного синтаксиса событий C# (на самом деле это стандартное поведение визуальных конструкторов Visual Studio). Когда вы обрабатываете события в такой манере, инфраструктура вызывает ваш специальный обработчик события после того, как реализация родительского класса завершена. Например, следующий код позволяет вручную обработать событие `MouseDown`:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    ...
    MouseDown += new MouseEventHandler(MainWindow_MouseDown);
  }
  private void MainWindow_MouseDown(object sender, MouseEventArgs e)
  {
    // Добавить код обработки события MouseDown.
  }
}
```

В дополнение к только что описанным методам `On`*`XXX`*`()` вы должны быть осведомлены о ряде других методов:

- `Hide()` — скрывает элемент управления и устанавливает свойство `Visible` в `false`;
- `Show()` — делает элемент управления видимым и устанавливает свойство `Visible` в `true`;
- `Invalidate()` — заставляет элемент управления перерисовать себя, отправив событие `Paint` (графическая визуализация с помощью GDI+ рассматривается позже в этом приложении).

Функциональность класса `Form`

Класс `Form` обычно (но не обязательно) является непосредственным базовым классом для специальных типов `Form`. Вдобавок к обширному набору членов, унаследованных от классов `Control`, `ScrollableControl` и `ContainerControl`, в типе `Form` определена дополнительная функциональность для главных окон, дочерних окон MDI и диалоговых окон. Давайте начнем с основных свойств, перечисленных в табл. А.5.

Таблица А.5. Свойства типа `Form`

Свойство	Описание
`AcceptButton`	Получает или устанавливает кнопочный элемент управления на форме, для которого имитируется щелчок при нажатии пользователем клавиши <Enter>

`ActiveMdiChild`
`IsMdiChild`
`IsMdiContainer`
	Используются внутри контекста приложений MDI

`CancelButton`	Получает или устанавливает кнопочный элемент управления на форме, для которого имитируется щелчок при нажатии пользователем клавиши <Esc>
`ControlBox`	Получает или устанавливает значение, которое указывает, имеется ли на форме область управления (т.е. значки для сворачивания, разворачивания и закрытия, находящиеся в правом верхнем углу окна)
`FormBorderStyle`	Получает или устанавливает стиль рамки формы. Применяется в сочетании с перечислением `FormBorderStyle`
`Menu`	Получает или устанавливает меню, стыкованное с формой
`MaximizeBox` `MinimizeBox`	Определяют, включены ли на форме значки развертывания и свертывания
`ShowInTaskbar`	Определяет, будет ли форма видна в панели задач Windows
`StartPosition`	Получает или устанавливает начальную позицию формы во время выполнения, указываемую посредством перечисления `FormStartPosition`
`WindowState`	Конфигурирует то, как форма отображается при запуске. Используется в сочетании с перечислением `FormWindowState`

Кроме многочисленных стандартных обработчиков событий с префиксом `On` в классе `Form` определены методы, основные из которых кратко описаны в табл. А.6.

Таблица А.6. Основные методы типа `Form`

Метод	Описание
`Activate()`	Активизирует заданную форму и предоставляет ей фокус ввода
`Close()`	Закрывает текущую форму
`CenterToScreen()`	Помещает форму в центр экрана
`LayoutMdi()`	Упорядочивает все дочерние формы (как указано перечислением `MdiLayout`) внутри родительской формы
`Show()`	Отображает форму как немодальное окно
`ShowDialog()`	Отображает форму как модальное диалоговое окно

Наконец, в классе `Form` определен набор событий, многие из которых инициируются на протяжении жизненного цикла формы (табл. А.7).

Таблица А.7. События типа `Form`

Событие	Описание
`Activated`	Это событие возникает всякий раз, когда форма *активизируется*, что означает получение ею фокуса на рабочем столе
`FormClosed` `FormClosing`	Эти события возникают непосредственно перед закрытием и сразу после закрытия формы
`Deactivate`	Это событие возникает всякий раз, когда форма *деактивизируется*, что означает утерю ею фокуса на рабочем столе
`Load`	Это событие возникает после того, как форма размещена в памяти, но пока еще не видна на экране
`MdiChildActive`	Это событие возникает, когда активизируется дочернее окно

Жизненный цикл типа `Form`

Если вам приходилось заниматься построением пользовательских интерфейсов с применением инструментальных наборов для создания графических пользовательских интерфейсов, таких как Java Swing, Mac OS X Cocoa или WPF, то вы знаете, что с *оконными типами* связаны многочисленные события, которые инициируются на протяжении их жизненного цикла. То же самое остается справедливым и для Windows Forms. Как вы уже видели, жизненный цикл формы начинается с вызова конструктора класса перед его передачей методу `Application.Run()`.

Когда объект размещен в управляемой куче, инфраструктура генерирует событие `Load`. В обработчике события `Load` можно настроить внешний вид объекта `Form`, подготовить содержащиеся в нем дочерние элементы управления (например, `ListBox` и `TreeView`) или выделить ресурсы, необходимые для работы формы (вроде подключений к базам данных и прокси для удаленных объектов).

После события `Load` инициируется событие `Activated`, когда форма получает фокус как активное окно на рабочем столе. Логической противоположностью

событию Activated является событие Deactivate, которое возникает при утере формой фокуса как активного окна. Несложно догадаться, что события Activated и Deactivate могут генерироваться много раз за время жизни заданного объекта Form по мере того, как пользователь переключается между активными окнами и приложениями.

Когда пользователь решает закрыть форму, возникают два события: FormClosing и FormClosed. Событие FormClosing инициируется первым и удобно для вывода конечному пользователю надоедливого (но полезного) сообщения с просьбой подтвердить закрытие приложения. Такой шаг дает пользователю шанс сохранить любые важные данные, прежде чем приложение прекратит работу.

Событие FormClosing работает в сочетании с делегатом FormClosing EventHandler. Если вы установите свойство FormClosingEventArgs.Cancel в true, то окно не будет уничтожено и просто возвратится к нормальному функционированию. В случае установки FormClosingEventArgs.Cancel в false генерируется событие FormClosed, и приложение Windows Forms прекращает работу, что приводит к выгрузке домена приложения и завершению процесса.

Приведенное далее изменение конструктора формы обеспечивает обработку событий Load, Activated, Deactivate, FormClosing и FormClosed (вспомните, что IDE-среда автоматически генерирует подходящий делегат и обработчик события при двукратном нажатии клавиши <Tab> после ввода символов +=):

```
public MainWindow()
{
  InitializeComponent();
  // Обработать разнообразные события времени жизни формы.
  FormClosing += new FormClosingEventHandler(MainWindow_Closing);
  Load += new EventHandler(MainWindow_Load);
  FormClosed += new FormClosedEventHandler(MainWindow_Closed);
  Activated += new EventHandler(MainWindow_Activated);
  Deactivate += new EventHandler(MainWindow_Deactivate);
}
```

Внутри обработчиков событий Load, FormClosed, Activated и Deactivate потребуется изменить значение новой строковой переменной-члена уровня Form (по имени lifeTimeInfo), указав простое сообщение, которое отображает имя перехваченного события. Добавьте следующую переменную-член в свой производный от Form класс:

```
public partial class MainWindow : Form
{
  private string lifeTimeInfo = "";
  ...
}
```

После этого реализуйте обработчики событий. В обработчике события FormClosed значение строки lifeTimeInfo отображается в диалоговом окне:

```
private void MainWindow_Load(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Load event\n";
}
```

```
private void MainWindow_Activated(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Activate event\n";
}
private void MainWindow_Deactivate(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Deactivate event\n";
}
private void MainWindow_Closed(object sender, FormClosedEventArgs e)
{
  lifeTimeInfo += "FormClosed event\n";
  MessageBox.Show(lifeTimeInfo);
}
```

Внутри обработчика события `FormClosing` пользователю выводится запрос, действительно ли он желает завершить работу приложения, с входными аргументами `FormClosingEventArgs`. В следующем коде метод `MessageBox.Show()` возвращает объект типа `DialogResult`, содержащий значение, которое представляет кнопку, выбранную конечным пользователем. Здесь диалоговое окно содержит кнопки Yes (Да) и No (Нет); таким образом, возвращаемое `Show()` значение проверяется на равенство `DialogResult.No`:

```
private void MainWindow_Closing(object sender, FormClosingEventArgs e)
{
  lifeTimeInfo += "FormClosing event\n";

  // Отобразить диалоговое окно с кнопками Yes и No.
  DialogResult dr = MessageBox.Show("Do you REALLY want to close this app?",
    "Closing event!", MessageBoxButtons.YesNo);

  // На какой кнопке пользователь выполнил щелчок?
  if (dr == DialogResult.No)
    e.Cancel = true;
  else
    e.Cancel = false;
}
```

Осталось внести еще одно изменение. В текущий момент выбор пункта меню File⇨Exit приводит к уничтожению всего приложения, что выглядит излишне радикальной мерой. Гораздо чаще в обработчике File⇨Exit окна верхнего уровня вызывается унаследованный метод `Close()`, который генерирует события, связанные с закрытием, и лишь затем объект приложения уничтожается:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Application.Exit();
  Close();
}
```

Запустите приложение и несколько раз переместите фокус на форму и обратно (для инициирования событий `Activated` и `Deactivate`). Когда вы, наконец, прекратите работу приложения, отобразится диалоговое окно, которое выглядит подобно показанному на рис. А.12.

Рис. А.12. Жизненный цикл производного от `Form` типа

Исходный код. Проект `SimpleVSWinFormsApp` находится в подкаталоге `Appendix_A`.

Реагирование на действия мыши и клавиатуры

Вспомните, что родительский класс `Control` определяет набор событий, которые позволяют отслеживать действия мыши и клавиатуры различными способами. Создайте новый проект Windows Forms Application по имени `MouseAndKeyboardEventsApp`, переименуйте (в окне Solution Explorer) форму в `MainWindow.cs` и обработайте событие `MouseMove` с использованием окна Properties. Результатом будет следующий обработчик события:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    InitializeComponent();
  }
  // Сгенерирован посредством окна Properties.
  private void MainWindow_MouseMove(object sender, MouseEventArgs e)
  {
  }
}
```

Событие `MouseMove` работает в сочетании с делегатом `System.Windows.Forms.MouseEventHandler`, который может вызывать только методы с первым параметром типа `System.Object` и вторым параметром типа `MouseEventArgs`. Этот тип содержит различные члены, которые предоставляют подробную информацию о состоянии события, связанного с мышью:

```
public class MouseEventArgs : EventArgs
{
  public MouseEventArgs(MouseButtons button, int clicks, int x,
    int y, int delta);
  public MouseButtons Button { get; }
  public int Clicks { get; }
  public int Delta { get; }
  public Point Location { get; }
  public int X { get; }
  public int Y { get; }
}
```

Большинство открытых свойств самоочевидны, а в табл. А.8 приведены дополнительные детали.

Таблица А.8. Свойства типа `MouseEventArgs`

Свойство	Описание
`Button`	Получает информацию о нажатой кнопке мыши, представленную с помощью перечисления `MouseButtons`
`Clicks`	Получает количество нажатия и отпускания кнопки мыши
`Delta`	Получает количество делений (которые соответствуют одному сдвигу колесика мыши) со знаком для текущего поворота колесика мыши
`Location`	Возвращает объект `Point`, содержащий координаты X и Y мыши
`X`	Получает координату X щелчка
`Y`	Получает координату Y щелчка

Наступило время реализовать обработчик события `MouseMove`, который будет отображать в заголовке `Form` текущие координаты X и Y мыши; для этого применяется свойство `Location`:

```
private void MainWindow_MouseMove(object sender, MouseEventArgs e)
{
  Text = string.Format("Mouse Position: {0}", e.Location);
}
```

Запустив приложение и перемещая курсор мыши по окну, вы обнаружите, что позиция курсора отображается в поле заголовка типа `MainWindow` (рис. А.13).

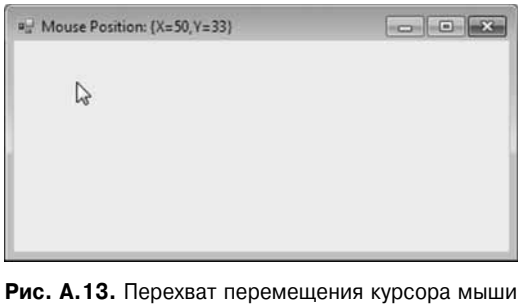

Рис. А.13. Перехват перемещения курсора мыши

Определение кнопки мыши, которой был выполнен щелчок

Еще одним действием, связанным с мышью, является выяснение, какой кнопкой был совершен щелчок при возникновении события MouseUp, MouseDown, MouseClick или MouseDoubleClick. Когда требуется точно знать, какая кнопка была задействована (левая, правая или средняя), необходимо проверить значение свойства Button класса MouseEventArgs. Значения свойства Button регламентируются перечислением MouseButtons:

```
public enum MouseButtons
{
  Left,
  Middle,
  None,
  Right,
  XButton1,
  XButton2
}
```

На заметку! Значения XButton1 и XButton2 позволяют перехватывать кнопки перехода вперед и назад, которые имеются у многих устройств, совместимых с контроллером мыши.

Все это можно увидеть в действии, обработав событие MouseDown типа MainWindow с использованием окна Properties. Следующий обработчик события MouseDown выводит в окне сообщения информацию о том, какой кнопкой мыши был произведен щелчок:

```
private void MainWindow_MouseDown (object sender, MouseEventArgs e)
{
  // Какой кнопкой был выполнен щелчок?
  if(e.Button == MouseButtons.Left)
    MessageBox.Show("Left click!");       // Щелчок левой кнопкой
  if(e.Button == MouseButtons.Right)
    MessageBox.Show("Right click!");      // Щелчок правой кнопкой
  if (e.Button == MouseButtons.Middle)
    MessageBox.Show("Middle click!");     // Щелчок средней кнопкой
}
```

Определение нажатой клавиши

В приложениях Windows обычно определены многочисленные элементы управления вводом (например, TextBox), в которых пользователь может вводить информацию с применением клавиатуры. Когда клавиатурный ввод перехватывается в подобной манере, нет необходимости в явной обработке событий клавиатуры, поскольку текстовую информацию можно извлекать из элемента управления, используя разнообразные свойства (например, Text в типе TextBox).

Тем не менее, если требуется отслеживать клавиатурный ввод для более экзотических целей (например, для фильтрации символов, поступающих в элемент управления, или захвата нажатий клавиш на самой форме), то библиотеки базовых классов предоставляют события KeyUp и KeyDown. Эти события работают в соче-

тании с делегатом KeyEventHandler, который может указывать на любой метод, принимающий тип object в первом параметре и тип KeyEventArgs — во втором. Вот определение типа KeyEventArgs:

```
public class KeyEventArgs : EventArgs
{
  public KeyEventArgs(Keys keyData);

  public virtual bool Alt { get; }
  public bool Control { get; }
  public bool Handled { get; set; }
  public Keys KeyCode { get; }
  public Keys KeyData { get; }
  public int KeyValue { get; }
  public Keys Modifiers { get; }
  public virtual bool Shift { get; }
  public bool SuppressKeyPress { get; set; }
}
```

В табл. А.9 кратко описаны некоторые наиболее интересные свойства, поддерживаемые типом KeyEventArgs.

Таблица А.9. Свойства типа KeyEventArgs

Свойство	Описание
Alt	Получает значение, которое указывает, была ли нажата клавиша <Alt>
Control	Получает значение, которое указывает, была ли нажата клавиша <Ctrl>
Handled	Получает или устанавливает значение, которое указывает, было ли событие полностью обработано в обработчике
KeyCode	Получает код клавиши для события KeyDown или KeyUp
Modifiers	Указывает, какие клавиши-модификаторы (<Ctrl>, <Shift> и/или <Alt>) были нажаты
Shift	Получает значение, которое указывает, была ли нажата клавиша <Shift>

Обработав событие KeyDown, свойства можно наблюдать в действии:

```
private void MainWindow_KeyDown(object sender, KeyEventArgs e)
{
  Text = string.Format("Key Pressed: {0} Modifiers: {1}",
                       e.KeyCode.ToString(), e.Modifiers.ToString());
}
```

Скомпилируйте и запустите приложение. Вы должны теперь видеть, какой кнопкой мыши был выполнен щелчок, а также какая клавиша была нажата. Например, на рис. А.14 показан результат одновременного нажатия клавиш <Ctrl> и <Shift>.

Исходный код. Проект MouseAndKeyboardEventsApp находится в подкаталоге Appendix_A.

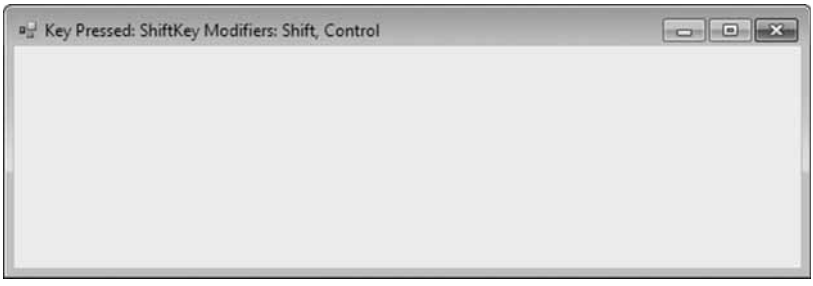

Рис. А.14. Перехват действий клавиатуры

Проектирование диалоговых окон

Внутри программы с графическим пользовательским интерфейсом диалоговые окна обычно являются главным способом захвата данных, вводимых пользователем, для применения в самом приложении. В отличие от других API-интерфейсов, которые вы могли использовать ранее, в Windows Forms отсутствует базовый класс вроде `Dialog`. Все диалоговые окна являются просто типами, производными от класса `Form`.

Кроме того, диалоговые окна задуманы как не поддерживающие изменение размеров, поэтому свойству `FormBorderStyle` обычно присваивается значение `FormBorderStyle.FixedDialog`. Вдобавок свойства `MinimizeBox` и `MaximizeBox` обычно устанавливаются в `false`. В итоге диалоговое окно конфигурируется как имеющее постоянные размеры. Наконец, если установить свойство `ShowInTaskbar` в `false`, то форма не будет видимой в панели задач Windows.

Давайте посмотрим, как строить и манипулировать диалоговыми окнами. Создайте новый проект Windows Forms Application по имени `CarOrderApp` и в окне Solution Explorer переименуйте первоначальный файл `Form1.cs` в `MainWindow.cs`. С помощью визуального конструктора форм создайте простое меню File⇨Exit (Файл⇨Выход), а также Tool⇨Order Automobile (Сервис⇨Заказ автомобиля). Вспомните, что меню строится путем перетаскивания элемента `MenuStrip` из панели инструментов на поверхность визуального конструктора с последующим конфигурированием пунктов меню. Обработайте события `Click` для пунктов меню Exit и Order Automobile с применением окна Properties.

Обработчик пункта меню File⇨Exit завершает работу приложения посредством вызова `Close()`:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
  Close();
}
```

В Visual Studio выберите в меню Project (Проект) пункт Add Windows Forms (Добавить форму Windows Forms) и назначьте новой форме имя `OrderAutoDialog.cs` (рис. А.15).

В рассматриваемом примере постройте диалоговое окно с традиционными кнопками OK и Cancel (именованными как `btnOK` и `btnCancel`), а также тремя полями `TextBox` с именами `txtMake`, `txtColor` и `txtPrice`.

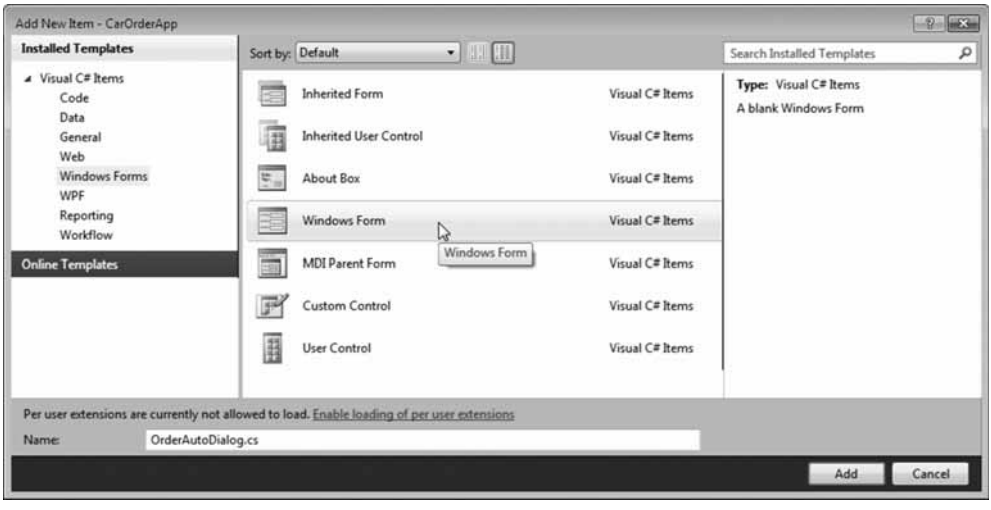

Рис. A.15. Вставка новых диалоговых окон в Visual Studio

Теперь воспользуйтесь окном Properties, чтобы завершить создание диалогового окна:

- установите свойство `FormBorderStyle` в `FixedDialog`;
- установите свойства `MinimizeBox` и `MaximizeBox` в `false`;
- установите свойство `StartPosition` в `CenterParent`;
- установите свойство `ShowInTaskbar` в `false`.

Свойство `DialogResult`

Выберите кнопку OK и в окне Properties установите свойство `DialogResult` в `OK`. Аналогичным образом установите свойство `DialogResult` кнопки Cancel в `Cancel`. Как вы вскоре увидите, свойство `DialogResult` довольно полезно тем, что позволяет быстро выяснить, на какой кнопке формы пользователь совершил щелчок, и предпринять соответствующее действие. Свойство `DialogResult` может быть установлено в любое значение из перечисления `DialogResult`:

```
public enum DialogResult
{
  Abort, Cancel, Ignore, No,
  None, OK, Retry, Yes
}
```

На рис. A.16 показан возможный дизайн диалогового окна; в нем даже предусмотрено несколько описательных меток.

Конфигурирование порядка обхода по нажатию клавиши <Tab>

Вы создали довольно привлекательное диалоговое окно; теперь необходимо формализовать порядок обхода по нажатию клавиши <Tab>. Вам наверняка известно, что многие пользователи ожидают наличия возможности перемещать фокус ввода с помощью клавиши <Tab>, когда форма содержит несколько графических элементов.

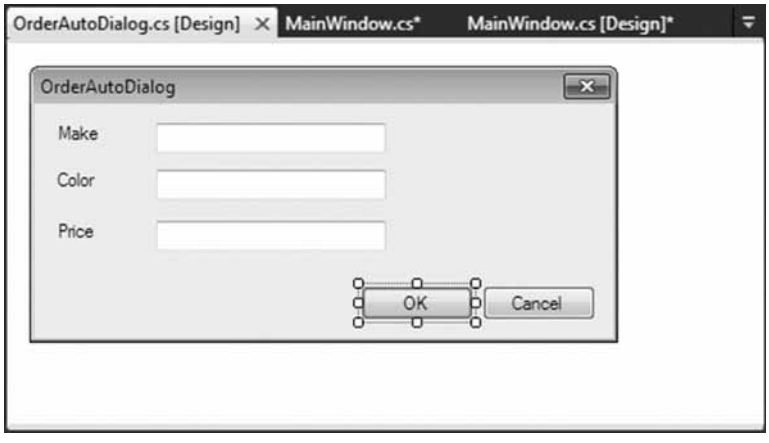

Рис. А.16. Тип `OrderAutoDialog`

Конфигурирование порядка обхода для набора элементов управления требует знания двух свойств: `TabStop` и `TabIndex`.

Свойство `TabStop` можно устанавливать в `true` или `false` на основе того, хотите ли вы, чтобы этот элемент графического пользовательского интерфейса был достижим с применением клавиши <Tab>. Предполагая, что свойство `TabStop` для заданного элемента управления установлено в `true`, вот как использовать свойство `TabOrder` для установления порядка активизации в последовательности переходов (нумерация в ней начинается с нуля):

```
// Настройка порядка обхода по нажатию клавиши <Tab>.
txtMake.TabIndex = 0;
txtMake.TabStop = true;
```

Мастер порядка обхода

Свойства `TabStop` и `TabIndex` можно устанавливать вручную в окне Properties; однако в IDE-среде Visual Studio предлагается мастер порядка обхода (Tab Order Wizard), который доступен через пункт меню View⇨Tab Order (Вид⇨Порядок обхода). Имейте в виду, что этот пункт меню виден только при активном визуальном конструкторе форм. После выбора пункта меню View⇨Tab Order для каждого графического элемента формы отобразится текущее значение `TabIndex`. Чтобы изменить эти значения, щелкайте на элементах в той очередности, в которой вы хотите выполнять обход по нажатию <Tab> (рис. А.17).

Установка стандартной кнопки ввода для формы

Во многих формах для пользовательского ввода (особенно в диалоговых окнах) предусмотрена специальная кнопка, которая автоматически реагирует на нажатие пользователем клавиши <Enter>. Предположим, что при нажатии пользователем клавиши <Enter> должен вызваться обработчик события `Click` кнопки `btnOK`.

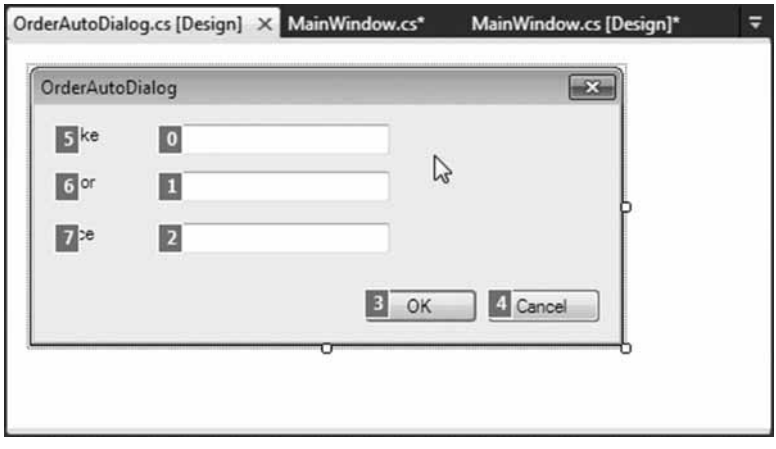

Рис. А.17. Мастер порядка обхода

Для этого нужно просто установить свойство `AcceptButton` в коде (или в окне Properties):

```
public partial class OrderAutoDialog : Form
{
  public OrderAutoDialog()
  {
    InitializeComponent();
    // Нажатие клавиши <Enter> вызывает ту же
    // реакцию, что и щелчок на кнопке btnOK.
    this.AcceptButton = btnOK;
  }
}
```

Завершить работу мастера порядка обхода можно нажатием клавиши <Esc>.

На заметку! Некоторые формы требуют возможности эмулирования щелчка на кнопке Cancel при нажатии пользователем клавиши <Esc>. Достичь этого можно присваиванием свойству `CancelButton` формы объекта `Button`, который представляет кнопку Cancel.

Отображение диалоговых окон

Планируя отображение диалогового окна, первым делом вы должны решить, в каком режиме его открывать: *модальном* или *немодальном*. Как вы, скорее всего, уже знаете, модальные диалоговые окна должны быть закрыты пользователем, прежде чем он сможет вернуться в окно, из которого первоначально было открыто диалоговое окно; например, большинство окон с информацией о программе являются модальными по своей природе. Чтобы отобразить новое диалоговое окно, вызовите метод `ShowDialog()` на объекте этого диалогового окна. С другой стороны, вы можете отобразить немодальное диалоговое окно, вызвав метод `Show()`, который позволяет пользователю переключать фокус между диалоговым окном и главным окном (подобно окну поиска/замены).

В текущем примере необходимо модифицировать обработчик пункта меню Tools⇨Order Automobile для типа `MainWindow`, чтобы объект `OrderAutoDialog` отображался в модальном режиме. Вот первоначальный код:

```
private void orderAutomobileToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Создать объект диалогового окна.
  OrderAutoDialog dlg = new OrderAutoDialog();

  // Отобразить как модальное диалоговое окно и выяснить, на какой кнопке
  // был выполнен щелчок, используя возвращаемое значение DialogResult.
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    // Пользователь щелкнул на кнопке OK; предпринять соответствующие действия...
  }
}
```

На заметку! Методы `ShowDialog()` и `Show()` можно вызывать с указанием объекта, который представляет владельца диалогового окна (для формы, загружающей диалоговое окно, это будет `this`). Указание владельца диалогового окна устанавливает z-упорядочение форм и также гарантирует (в случае немодального диалогового окна), что при уничтожении главного окна будут освобождены и все принадлежащие ему окна.

Имейте в виду, что при создании экземпляра производного от `Form` типа (в данном случае `OrderAutoDialog`) окно *не* отображается на экране, а лишь размещается в памяти. Видимым оно становится только после вызова метода `Show()` или `ShowDialog()`. Кроме того, обратите внимание, что метод `ShowDialog()` возвращает значение `DialogResult`, которое было присвоено одной из кнопок (метод `Show()` возвращает `void`).

После завершения метода `ShowDialog()` форма больше не видна на экране, но по-прежнему находится в памяти. Это означает, что можно извлечь значения из каждого элемента `TextBox`. Тем не менее, попытка компиляции следующего кода приведет к выдаче сообщения об ошибке:

```
private void orderAutomobileToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Создать объект диалогового окна.
  OrderAutoDialog dlg = new OrderAutoDialog();

  // Отобразить как модальное диалоговое окно и выяснить, на какой кнопке
  // был выполнен щелчок, используя возвращаемое значение DialogResult.
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    // Получить значения из текстовых полей? Ошибки на этапе компиляции!
    string orderInfo = string.Format("Марка: {0}, Цвет: {1}, Цена: {2}",
      dlg.txtMake.Text, dlg.txtColor.Text, dlg.txtPrice.Text);
    MessageBox.Show(orderInfo, "Information about your order!");
  }
}
```

Ошибка на этапе компиляции возникает из-за того, что Visual Studio объявляет элементы управления, добавляемые на поверхность визуального конструктора

форм, как *закрытые* переменные-члены класса. Можете удостовериться в этом, открыв файл OrderAutoDialog.Designer.cs.

Реальный класс диалогового окна мог бы поддерживать инкапсуляцию за счет добавления открытых свойств для установки и получения значений из текстовых полей, но можно поступить проще и переопределить их как public. Выберите в конструкторе каждый элемент TextBox и установите его свойство Modifiers в Public с применением окна Properties. Это изменит код для визуального конструктора:

```
partial class OrderAutoDialog
{
  ...
  // Переменные-члены формы определены в файле,
  // который сопровождается конструктором.
  public System.Windows.Forms.TextBox txtMake;
  public System.Windows.Forms.TextBox txtColor;
  public System.Windows.Forms.TextBox txtPrice;
}
```

Теперь можете скомпилировать и запустить приложение. После отображения диалогового окна и щелчка на кнопке OK вы увидите в окне сообщения данные, введенные в полях.

Наследование форм

До сих пор все специальные и диалоговые окна, рассмотренные в приложении, были порождены непосредственно от класса System.Windows.Forms.Form. Однако интересным аспектом разработки приложений Windows Forms является тот факт, что типы Form могут выступать в качестве базовых классов для производных типов Form. Предположим, что вы создали библиотеку кода .NET, которая содержит все основные диалоговые окна компании. Позже вы решили, что в окно "О программе" неплохо было бы добавить трехмерный логотип компании. Вместо пересоздания всего окна вы можете расширить базовое окно "О программе", унаследовав его внешний вид и поведение:

```
// ThreeDAboutBox "является" AboutBox
public partial class ThreeDAboutBox : AboutBox
{
  // Код отображения логотипа компании...
}
```

Чтобы увидеть наследование форм в действии, вставьте в свой проект новую форму, используя пункт меню Project⇨Add Windows Form (Проект⇨Добавить форму Windows Forms), но на этот раз выберите вариант Inherited Form (Унаследованная форма) и назовите новую форму ImageOrderAutoDialog.cs (рис. A.18).

При выборе этого варианта открывается диалоговое окно Inheritance Picker (Выбор наследования), которое отображает все формы в текущем проекте. Обратите внимание, что кнопка Browse (Обзор) позволяет выбрать форму из внешней сборки .NET. В рассматриваемом примере просто выберите класс OrderAutoDialog.

На заметку! Чтобы видеть формы проекта в диалоговом окне Inheritance Picker, вы должны хотя бы раз скомпилировать проект, т.к. для отображения доступных вариантов этот инструмент читает метаданные сборки.

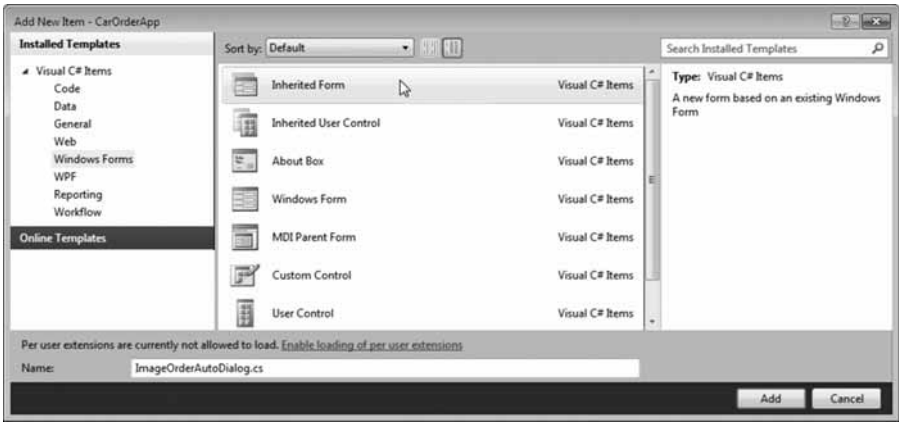

Рис. А.18. Добавление в проект производной формы

После щелчка на кнопке OK инструменты визуального проектирования отобразят базовые элементы управления на производных элементах; каждый элемент управления имеет слева вверху небольшой значок со стрелкой, обозначающий наследование. Чтобы завершить построение производного диалогового окна, найдите элемент управления `PictureBox` в разделе Common Controls (Общие элементы управления) панели инструментов и добавьте его на производную форму. Затем с помощью свойства `Image` выберите подходящий файл с изображением. На рис. А.19 показан возможный вид пользовательских интерфейсов.

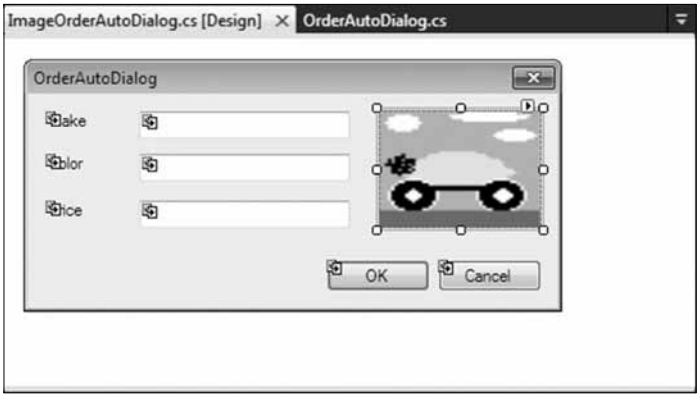

Рис. А.19. Пользовательский интерфейс класса `ImageOrderAutoDialog`

Теперь вы можете модифицировать обработчик события `Click` для пункта меню Tools⇨Order Automobile, чтобы создавать экземпляр производного типа, а не базового класса `OrderAutoDialog`:

```
private void orderAutomobileToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Создать объект производного типа диалогового окна.
  ImageOrderAutoDialog dlg = new ImageOrderAutoDialog();
  ...
}
```

Исходный код. Проект `CarOrderApp` находится в подкаталоге `Appendix_A`.

Визуализация графических данных с использованием GDI+

Многие приложения с графическим пользовательским интерфейсом требуют возможности динамической генерации графических данных для отображения на поверхности окна. Пусть, например, вы извлекли из реляционной базы данных набор записей и хотите визуализировать круговую (или столбчатую) диаграмму, которая наглядно показывает наличие товаров на складе. Или, скажем, вам нужно воссоздать старую видеоигру с применением платформы .NET. Безотносительно к цели GDI+ является API-интерфейсом, который должен использоваться для визуализации графических данных в приложениях Windows Forms. Эта технология сконцентрирована в сборке `System.Drawing.dll`, которая определяет несколько пространств имен (рис. A.20).

На заметку! Не забывайте, что инфраструктура WPF располагает собственной подсистемой графической визуализации и API-интерфейсом; вы будете применять GDI+ только внутри приложений Windows Forms.

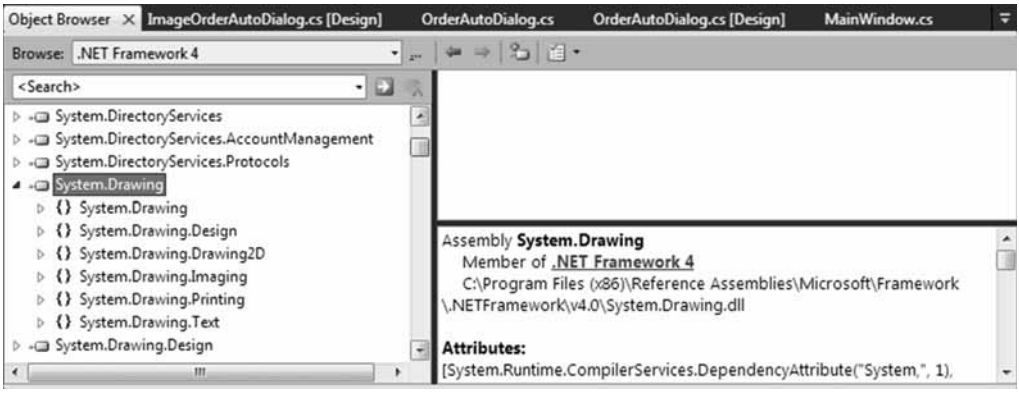

Рис. A.20. Пространства имен сборки `System.Drawing.dll`

В табл. A.10 приведены высокоуровневые описания основных пространств имен GDI+.

Пространство имен `System.Drawing`

Подавляющее большинство типов, которые вы будете применять при разработке приложений GDI+, находятся в пространстве имен `System.Drawing`. Вполне ожидаемо, в нем можно обнаружить классы, которые представляют изображения, кисти, перья и шрифты. Кроме того, в `System.Drawing` определено несколько связанных вспомогательных типов, таких как `Color`, `Point` и `Rectangle`.

Таблица А.10. Основные пространства имен GDI+

Пространство имен	Описание
`System.Drawing`	Основное пространство имен GDI+, в котором определены многочисленные типы для базового отображения (скажем, шрифты, перья и простые кисти), а также мощный тип `Graphics`
`System.Drawing.Drawing2D`	Это пространство имен предоставляет типы, используемые для более сложной функциональности двумерной/векторной графики (например, градиентные кисти, кончики пера и геометрические трансформации)
`System.Drawing.Imaging`	В этом пространстве определены типы, которые позволяют манипулировать графическими изображениями (скажем, изменять палитру, извлекать метаданные изображения, управлять метафайлами и т.д.)
`System.Drawing.Printing`	В этом пространстве определены типы, позволяющие визуализировать изображения на печатных страницах, взаимодействовать с принтерами и форматировать общий внешний вид для печатного задания
`System.Drawing.Text`	Это пространство позволяет манипулировать коллекциями шрифтов

В табл. А.11 перечислены некоторые (но не все) основные типы.

Таблица А.11. Основные типы пространства имен `System.Drawing`

Тип	Описание
`Bitmap`	Инкапсулирует данные изображения (`*.bmp` или другого)

`Brush`
`Brushes`
`SolidBrush`
`SystemBrushes`
`TextureBrush`
	Объекты кистей используются для заполнения внутренностей графических фигур, таких как прямоугольники, эллипсы и многоугольники

`BufferedGraphics`	Предоставляет графический буфер для двойной буферизации, которая применяется для сокращения или устранения мерцания вследствие перерисовки поверхности элемента
`Color` `SystemColors`	Определяют несколько статических свойств только для чтения, которые используются для получения специфических цветов для конструирования разнообразных перьев и/или кистей
`Font` `FontFamily`	Тип `Font` инкапсулирует характеристики заданного шрифта (имя, плотность, наклон и размер в пунктах). Тип `FontFamily` предоставляет абстракцию для группы шрифтов похожего вида, но с некоторыми расхождениями в стиле
`Graphics`	Основной класс, представляющий допустимую поверхность рисования, а также несколько методов для визуализации текста, изображений и геометрических фигур
`Icon` `SystemIcons`	Представляют специальные значки, а также набор стандартных значков, предлагаемых системой
`Image` `ImageAnimator`	`Image` — абстрактный базовый класс, который предоставляет функциональность типам `Bitmap`, `Icon` и `Cursor`. Класс `ImageAnimator` позволяет повторять несколько производных от `Image` типов через указанный интервал времени

Окончание табл. А.11

Тип	Описание

`Pen`
`Pens`
`SystemPens`
	Перья — это объекты, применяемые для рисования прямых и кривых линий. В типе `Pens` определено несколько статических свойств, которые возвращают новый объект `Pen` с заданным цветом

`Point` `PointF`	Структуры, представляющие отображение координат (X, Y) на лежащие в основе целые и дробные значения
`Rectangle` `RectangleF`	Структуры, представляющие размерность прямоугольника (также отображаются на целые и дробные значения)
`Size` `SizeF`	Структуры, представляющие высоту и ширину (также отображаются на целые и дробные значения)
`StringFormat`	Инкапсулирует разнообразные характеристики текстовой компоновки (например, выравнивание и межстрочный интервал)
`Region`	Описывает внутреннюю область геометрической фигуры, образованной из прямоугольников и ломаных линий

Роль типа `Graphics`

Класс `System.Drawing.Graphics` служит в качестве шлюза для функциональности визуализации GDI+. Он не только представляет поверхность для рисования (такую как поверхность формы, поверхность элемента управления или область памяти), но и определяет десятки членов, которые позволяют визуализировать текст, рисунки (вроде значков и растровых изображений) и многочисленные геометрические фигуры. Неполный список членов класса приведен в табл. А.12.

Таблица А.12. Избранные члены класса `Graphics`

Член	Описание

`FromHdc()`
`FromHwnd()`
`FromImage()`
	Статические методы, предоставляющие способ получения объекта `Graphics` из заданного изображения (например, значка или растрового изображения) или элемента управления графического пользовательского интерфейса

`Clear()`	Заполняет объект `Graphics` указанным цветом, в процессе стирая текущую поверхность рисования

`DrawArc()`
`DrawBeziers()`
`DrawCurve()`
`DrawEllipse()`
`DrawIcon()`
`DrawLine()`
`DrawLines()`
`DrawPie()`
`DrawPath()`
`DrawRectangle()`
`DrawRectangles()`
`DrawString()`
	Методы для визуализации заданного изображения или геометрической фигуры. Все методы `DrawXXX()` требуют применения объектов `Pen` из GDI+

`FillEllipse()`
`FillPie()`
`FillPolygon()`
`FillRectangle()`
`FillPath()`
	Методы для заполнения внутренней области заданной геометрической фигуры. Все методы `FillXXX()` требуют использования объектов `Brush` из GDI+

Обратите внимание, что экземпляры класса Graphics невозможно создавать напрямую с применением ключевого слова new, т.к. открытые конструкторы отсутствуют. Как тогда получить объект Graphics? Ищите ответ ниже.

Получение объекта Graphics с помощью события Paint

Объект Graphics чаще всего получают с использованием окна Properties в Visual Studio для обработки события Paint окна, где необходимо выполнить визуализацию. Это событие определено посредством делегата PaintEventHandler, который может указывать на любой метод, принимающий System.Object в первом параметре и PaintEventArgs — во втором.

Параметр PaintEventArgs содержит объект Graphics, который нужно визуализировать на поверхности формы. В качестве примера создайте новый проект Windows Forms Application по имени PaintEventApp. В окне Solution Explorer измените имя первоначального файла Form.cs на MainWindow.cs и в окне Properties создайте обработчик события Paint. В итоге создается следующий код заглушки:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    InitializeComponent();
  }
  private void MainWindow_Paint(object sender, PaintEventArgs e)
  {
    // Добавьте сюда свой код рисования.
  }
}
```

Теперь, имея обработчик события Paint, может возникнуть вопрос: когда инициируется это событие? Событие Paint возникает всякий раз, когда окно становится "запорченным" из-за того, что изменены его размеры, перестает загораживаться (частично или полностью) другим окном или разворачивается после сворачивания. Во всех этих случаях платформа .NET обеспечивает автоматический вызов обработчика события Paint. Взгляните на показанную ниже реализацию обработчика MainWindow_Paint():

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
  // Получить объект Graphics для данной формы.
  Graphics g = e.Graphics;

  // Нарисовать окружность.
  g.FillEllipse(Brushes.Blue, 10, 20, 150, 80);
  // Нарисовать строку с заданным шрифтом.
  g.DrawString("Hello GDI+", new Font("Times New Roman", 30),
               Brushes.Red, 200, 200);

  // Нарисовать линию с заданным пером.
  using (Pen p = new Pen(Color.YellowGreen, 10))
  {
    g.DrawLine(p, 80, 4, 200, 200);
  }
}
```

После получения объекта `Graphics` из входного параметра `PaintEventArgs` вызывается метод `FillEllipse()`. Этот метод (как и любой метод с именем, начинающимся на `Fill`) требует передачи в первом параметре объекта производного от `Brush` типа. Хотя можно было бы создать любое количество интересных объектов кистей с применением классов из пространства имен `System.Drawing.Drawing2D` (вроде `HatchBrush` и `LinearGradientBrush`), служебный класс `Brushes` предлагает удобный доступ к разнообразным типам кистей со сплошными цветами.

Затем вызывается метод `DrawString()`, которому в первом параметре передается визуализируемая строка. Для этого в GDI+ предусмотрен тип `Font`, представляющий не только имя шрифта для визуализации текстовых данных, но и характеристики шрифта, такие как размер в пунктах (в данном случае 30). Обратите внимание, что методу `DrawString()` также требуется объект типа `Brush`; с точки зрения GDI+ строка `"Hello GDI+"` является не более чем набором геометрических фигур, которые необходимо вывести на экран. Наконец, вызывается метод `DrawLine()`, который визуализирует прямую линию, используя специальный тип `Pen` шириной 10 пикселей. Вывод логики визуализации показан на рис. А.21.

Рис. А.21. Простые операции визуализации GDI+

На заметку! В предыдущем коде объект `Pen` освобождается явно. Как правило, когда объект типа GDI+, реализующего интерфейс `IDisposable`, создается напрямую, после окончания работы с ним понадобится вызвать метод `Dispose()`. Это позволит освободить занятые ресурсы как можно раньше. Если не поступать так, то ресурсы в конечном итоге будут освобождены сборщиком мусора, но в недетерминированной манере.

Объявление недействительной клиентской области формы

Во время работы приложения Windows Forms может возникнуть необходимость явно сгенерировать в коде событие `Paint`, не ожидая, пока окно станет *естественно запорченным* в результате действий пользователя. Скажем, программа может предоставлять пользователю на выбор несколько заготовленных изображений с помощью специального диалогового окна. После закрытия диалогового окна выбранное изображение необходимо вывести в клиентской области формы. Очевидно, если ожидать естественного загрязнения формы, то пользователь не увидит никаких изменений, пока не изменит размер окна или не перекроет его другим окном. Чтобы программно выполнить перерисовку окна, понадобится вызвать унаследованный метод `Invalidate()`:

```
public partial class MainForm: Form
{
  ...
  private void MainForm_Paint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    // Визуализировать здесь корректное изображение.
  }
  private void GetImageFromDialog()
  {
    // Отобразить диалоговое окно и получить новое изображение.
    // Перерисовать всю клиентскую область.
    Invalidate();
  }
}
```

Метод `Invalidate()` имеет несколько перегруженных версий, которые позволяют указывать конкретную прямоугольную часть формы, подлежащую перерисовке, вместо того, чтобы перерисовывать всю клиентскую область (как происходит по умолчанию). Если требуется обновить только небольшой прямоугольник в верхнем левом углу клиентской области, то можно написать такой код:

```
// Перерисовать заданную прямоугольную область формы.
private void UpdateUpperArea()
{
  Rectangle myRect = new Rectangle(0, 0, 75, 150);
  Invalidate(myRect);
}
```

Исходный код. Проект `PaintEventApp` находится в подкаталоге `Appendix_A`.

Построение полного приложения Windows Forms

В заключение обзора API-интерфейсов Windows Forms и GDI+ мы построим полное приложение с графическим пользовательским интерфейсом, в котором будут продемонстрированы многие описанные здесь приемы. Это будет несложная программа для рисования, которая позволяет пользователям выбрать одну из двух фигур (круг или прямоугольник) и цвет для ее отображения на форме. Конечные пользователи смогут также сохранять свои рисунки в локальном файле на жестком диске с помощью служб сериализации объектов.

Создание системы главного меню

Начните с создания нового проекта Windows Forms Application по имени `MyPaintProgram` и измените первоначальное имя файла `Form1.cs` на `MainWindow.cs`. Затем постройте в пустом окне систему меню с пунктом File (Файл), в котором имеются подпункты Save... (Сохранить), Load... (Загрузить) и Exit (Выход), как показано на рис. А.22.

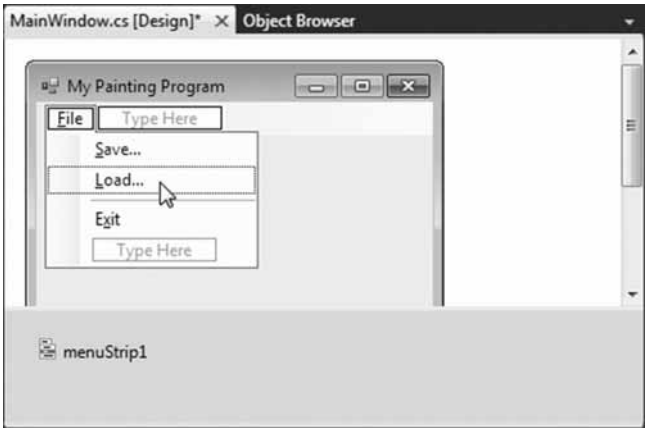

Рис. А.22. Система меню File

На заметку! Указание в качестве пункта меню одиночного дефиса (–) позволяет определять разделители в системе меню.

После этого создайте еще один пункт меню верхнего уровня Tools (Сервис), который содержит подпункты для выбора визуализируемой фигуры и ее цвета, а также для очистки формы ото всех графических данных (рис. А.23).

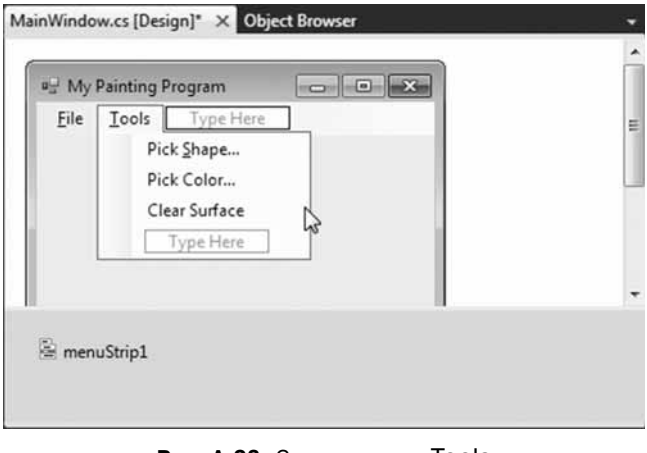

Рис. А.23. Система меню Tools

Наконец, обработайте событие `Click` для каждого из построенных подпунктов меню. Обработчики будут создаваться по мере выполнения примера, а пока реализуйте обработчик для пункта File⇨Exit, вызвав в нем метод `Close()`:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
  Close();
}
```

Определение типа ShapeData

Вспомните, что разрабатываемое приложение должно позволять конечным пользователям выбирать одну из двух предопределенных фигур заданного цвета. Кроме того, конечный пользователь должен иметь возможность сохранять свои графические данные в файле, поэтому нужно определить специальный класс, который инкапсулирует все необходимые детали; для простоты будут применяться автоматические свойства C#. Добавьте в проект новый файл класса ShapeData.cs со следующей реализацией:

```
[Serializable]
class ShapeData
{
  // Верхняя левая координата фигуры.
  public Point UpperLeftPoint { get; set; }

  // Текущий цвет фигуры.
  public Color Color { get; set; }

  // Вид фигуры.
  public SelectedShape ShapeType { get; set; }
}
```

В классе ShapeData используются три автоматических свойства, инкапсулирующие различные типы данных, два из которых (Point и Color) определены в пространстве имен System.Drawing, так что не забудьте импортировать упомянутое пространство имен в файл кода. Обратите внимание на наличие у класса ShapeData атрибута [Serializable]. Далее вы сконфигурируете тип MainWindow для поддержки списка объектов ShapeData, который будет сохраняться посредством служб сериализации объектов.

Определение типа ShapePickerDialog

Чтобы пользователь мог выбирать круг или прямоугольник, можно создать простое диалоговое окно по имени ShapePickerDialog (вставьте этот новый объект Form). Помимо традиционных кнопок OK и Cancel (каждой из которых должно быть присвоено подходящее значение DialogResult) в диалоговом окне будет находиться элемент GroupBox, содержащий два объекта RadioButton: radioButtonCircle и radioButtonRect. На рис. А.24 показан возможный дизайн.

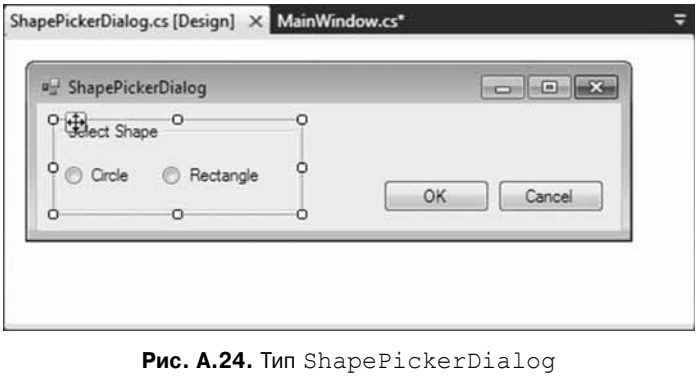

Рис. А.24. Тип ShapePickerDialog

Откройте редактор кода для данного диалогового окна, щелкнув правой кнопкой мыши на поверхности визуального конструктора форм и выбрав в контекстном меню пункт View Code (Просмотреть код). В пространстве имен `MyPaintProgram` определите перечисление `SelectedShape` с именами возможных фигур:

```
public enum SelectedShape
{
  Circle, Rectangle
}
```

Модифицируйте текущий класс `ShapePickerDialog`, как описано ниже.

- Добавьте автоматическое свойство типа `SelectedShape`. Вызывающий метод может применять это свойство для определения, какую фигуру визуализировать.
- Создайте обработчик события `Click` для кнопки OK, используя окно Properties.
- Реализуйте этот обработчик события, чтобы он определял, выбран ли переключатель `radioButtonCircle` (с помощью свойства `Checked`). Если он выбран, то установите свойство `SelectedShape` в `SelectedShape.Circle`, а иначе — в `SelectedShape.Rectangle`.

Вот полный код:

```
public partial class ShapePickerDialog : Form
{
  public SelectedShape SelectedShape { get; set; }
  public ShapePickerDialog()
  {
    InitializeComponent();
  }
  private void btnOK_Click(object sender, EventArgs e)
  {
    if (radioButtonCircle.Checked)
      SelectedShape = SelectedShape.Circle;
    else
      SelectedShape = SelectedShape.Rectangle;
  }
}
```

На этом инфраструктура программы завершена. Понадобится еще реализовать обработчики событий `Click` для остальных пунктов меню главного окна.

Добавление инфраструктуры в тип `MainWindow`

Возвратитесь к построению главного окна и добавьте к форме три новых переменных-члена. Эти переменные-члены позволят отслеживать выбранную фигуру (с помощью перечисления `SelectedShape`), выбранный цвет (представленный типом `System.Drawing.Color`) и все визуализированные изображения, содержащиеся в обобщенном списке `List<T>` (где `T` — тип `ShapeData`):

```
public partial class MainWindow : Form
{
  // Текущая фигура и цвет для визуализации.
  private SelectedShape currentShape;
  private Color currentColor = Color.DarkBlue;
```

```
  // Здесь содержатся все объекты ShapeData.
  private List<ShapeData> shapes = new List<ShapeData>();
  ...
}
```

Далее обработайте события `MouseDown` и `Paint` для данного производного от `Form` типа с применением окна Properties. Они будут реализованы на следующем шаге, а пока вы должны заметить, что IDE-среда сгенерировала заглушки:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
}

private void MainWindow_MouseDown(object sender, MouseEventArgs e)
{
}
```

Реализация функциональности меню Tools

Чтобы позволить пользователю установить значение переменной-члена `currentShape`, необходимо реализовать обработчик события `Click` для пункта меню Tools⇨Pick Shape... (Сервис⇨Выбрать фигуру...). Он должен открывать специальное диалоговое окно и на основе выбора пользователя присваивать переменной-члену соответствующее значение:

```
private void pickShapeToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Загрузка диалогового окна и установка корректного типа фигуры.
  ShapePickerDialog dlg = new ShapePickerDialog();
  if (DialogResult.OK == dlg.ShowDialog())
  {
    currentShape = dlg.SelectedShape;
  }
}
```

Чтобы предоставить пользователю возможность установки значения переменной-члена `currentColor`, понадобится реализовать обработчик события `Click` для пункта меню Tools⇨Pick Color... (Сервис⇨Выбрать цвет...), в котором используется тип `System.Windows.Forms.ColorDialog`:

```
private void pickColorToolStripMenuItem_Click(object sender, EventArgs e)
{
  ColorDialog dlg = new ColorDialog();
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    currentColor = dlg.Color;
  }
}
```

Запустив программу и выбрав пункт меню Tools⇨Pick Color..., вы увидите диалоговое окно, показанное на рис. А.25.

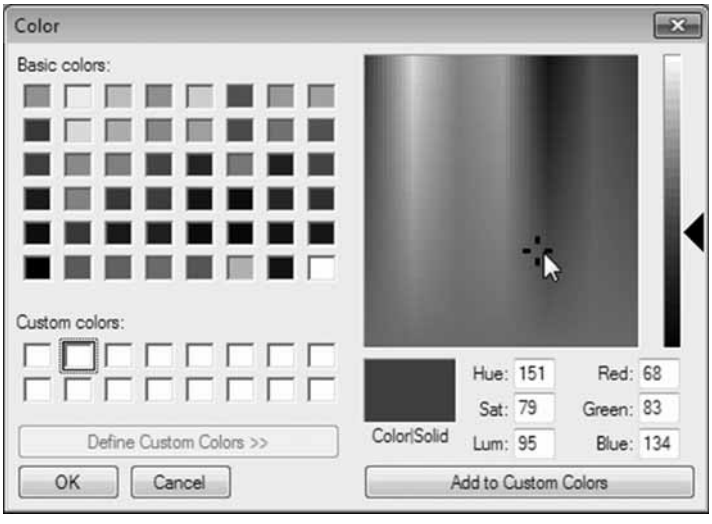

Рис. A.25. Стандартный тип `ColorDialog`

В завершение необходимо реализовать обработчик пункта меню Tools⇨Clear Surface, который очищает содержимое переменной-члена `List<T>` и программно инициирует событие `Paint` посредством вызова `Invalidate()`:

```
private void clearSurfaceToolStripMenuItem_Click(object sender, EventArgs e)
{
  shapes.Clear();
  // Инициирует событие Paint.
  Invalidate();
}
```

Захват и визуализация графического вывода

Поскольку вызов `Invalidate()` инициирует событие `Paint`, вы должны реализовать обработчик события `Paint`. Цель заключается в организации цикла по всем элементам (в настоящий момент пустой) переменной-члена типа `List<T>` и визуализировать в текущей позиции курсора мыши окружность или прямоугольник. Первым делом реализуйте обработчик события `MouseDown` и вставьте новый объект `ShapeData` в обобщенный список `List<T>`, учитывая выбранные пользователем цвет, вид и текущую позицию курсора мыши:

```
private void MainWindow_MouseDown(object sender, MouseEventArgs e)
{
  // Создать объект ShapeData на основе текущего пользовательского выбора.
  ShapeData sd = new ShapeData();
  sd.ShapeType = currentShape;
  sd.Color = currentColor;
  sd.UpperLeftPoint = new Point(e.X, e.Y);
  // Добавить в List<T> и принудительно перерисовать форму.
  shapes.Add(sd);
  Invalidate();
}
```

Далее реализуйте обработчик события Paint:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
  // Получить объект Graphics для текущего окна.
  Graphics g = e.Graphics;

  // Визуализировать каждую фигуру заданным цветом.
  foreach (ShapeData s in shapes)
  {
    // Визуализировать квадрат или круг размером 20x20 пикселей,
    // используя подходящий цвет.
    if (s.ShapeType == SelectedShape.Rectangle)
      g.FillRectangle(new SolidBrush(s.Color),
        s.UpperLeftPoint.X, s.UpperLeftPoint.Y, 20, 20);
    else
      g.FillEllipse(new SolidBrush(s.Color),
        s.UpperLeftPoint.X, s.UpperLeftPoint.Y, 20, 20);
  }
}
```

Если теперь запустить приложение, то должна появиться возможность визуализации любого количества фигур произвольных цветов (рис. А.26).

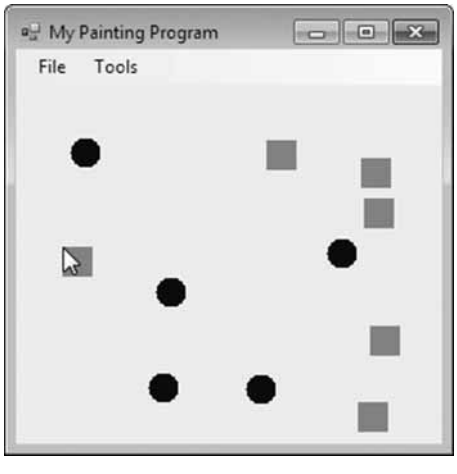

Рис. А.26. Приложение MyPaintProgram в действии

Реализация логики сериализации

Финальный аспект проект предусматривает реализацию обработчиков событий для пунктов меню File⇨Save... и File⇨Load.... С учетом того, что класс ShapeData снабжен атрибутом [Serializable] (и сам класс List<T> является сериализируемым), текущие графические данные можно сохранить с применением типа SaveFileDialog из Windows Forms. Начните с добавления директив using для пространств имен System.Runtime.Serialization.Formatters.Binary и System.IO:

```
// Для двоичного форматера.
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```

Затем модифицируйте обработчик события `Click` для пункта меню File⇨Save...:

```
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
  using (SaveFileDialog saveDlg = new SaveFileDialog())
  {
    // Сконфигурировать внешний вид диалогового окна сохранения файла.
    saveDlg.InitialDirectory = ".";
    saveDlg.Filter = "Shape files (*.shapes)|*.shapes";
    saveDlg.RestoreDirectory = true;
    saveDlg.FileName = "MyShapes";

    // Если пользователь щелкнул на кнопке OK, то
    // открыть новый файл и сериализировать List<T>.
    if (saveDlg.ShowDialog() == DialogResult.OK)
    {
      Stream myStream = saveDlg.OpenFile();
      if ((myStream != null))
      {
        // Сохранить фигуры.
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        myBinaryFormat.Serialize(myStream, shapes);
        myStream.Close();
      }
    }
  }
}
```

Обработчик события `Click` для пункта меню File⇨Load... открывает выбранный в диалоговом окне `OpenFileDialog` файл и десериализирует данные в переменную-член `List<T>`:

```
private void loadToolStripMenuItem_Click(object sender, EventArgs e)
{
  using (OpenFileDialog openDlg = new OpenFileDialog())
  {
    openDlg.InitialDirectory = ".";
    openDlg.Filter = "Shape files (*.shapes)|*.shapes";
    openDlg.RestoreDirectory = true;
    openDlg.FileName = "MyShapes";

    if (openDlg.ShowDialog() == DialogResult.OK)
    {
      Stream myStream = openDlg.OpenFile();
      if ((myStream != null))
      {
        // Получить фигуры.
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        shapes = (List<ShapeData>)myBinaryFormat.Deserialize(myStream);
```

```
            myStream.Close();
            Invalidate();
          }
        }
      }
    }
```

Общая логика сериализации должна выглядеть знакомой. Полезно отметить, что типы `SaveFileDialog` и `OpenFileDialog` поддерживают свойство `Filter`, которому можно присваивать строковое значение фильтра. Фильтр управляет несколькими настройками для диалоговых окон сохранения и открытия файлов, такими как файловое расширение (`*.shapes`). Свойство `FileName` используется для управления стандартным именем сохраняемого файла (`MyShapes` в рассмотренном примере).

На этом пример приложения для рисования фигур завершен. Вы должны иметь возможность сохранять графические данные в файлах `*.shapes` и впоследствии загружать их. Если вы хотите усовершенствовать приложение, то можете добавить дополнительные фигуры или даже позволить пользователю управлять размером и формой фигур, а также выбирать формат сохраняемых данных (двоичный, XML или SOAP).

Резюме

В настоящем приложении исследовался процесс построения традиционных настольных приложений с применением API-интерфейсов Windows Forms и GDI+, которые являются частью платформы .NET Framework, начиная с версии 1.0. Приложение Windows Forms состоит минимум из типа, расширяющего класс `Form`, и метода `Main()`, который взаимодействует с типом `Application`.

Заполнение формы элементами пользовательского интерфейса (скажем, системами меню и элементами управления вводом) осуществляется вставкой новых объектов в унаследованную коллекцию `Controls`. Здесь также было показано, как перехватывать события мыши, клавиатуры и визуализации. Попутно вы ознакомились с типом `Graphics` и множеством способов генерации графических данных во время выполнения.

Как упоминалось ранее, API-интерфейс Windows Forms (в некоторой степени) был вытеснен API-интерфейсом WPF, который появился в версии .NET 3.0. Хотя и верно то, что инфраструктура WPF удобна для создания насыщенных пользовательских интерфейсов, API-интерфейс Windows Forms остается простейшим (и во многих случаях наиболее прямым) способом построения стандартных бизнес-приложений, приложений для внутреннего потребления и простых утилит конфигурирования. По этим причинам в ближайшие годы инфраструктура Windows Forms продолжит быть частью библиотек базовых классов .NET.