ПРИЛОЖЕНИЕ A

Программирование с помощью Windows Forms

Со времени первоначального выхода платформы .NET (примерно в 2001 г.) среди библиотек базовых классов появился API-интерфейс по имени Windows Forms, представленный в основном сборкой `System.Windows.Forms.dll`. Инструментальный набор Windows Forms предоставляет типы, необходимые для построения графических пользовательских интерфейсов для настольных компьютеров, создания специальных элементов управления, управления ресурсами (например, таблицами строк и значками) и решения других задач, возникающих при программировании для настольных компьютеров. Имеется и дополнительный API-интерфейс по имени GDI+ (представленный сборкой `System.Drawing.dll`), который предоставляет дополнительные типы, позволяющие программисту генерировать двухмерную графику, взаимодействовать с сетевыми принтерами и манипулировать графическими данными.

API-интерфейсы Windows Forms (и GDI+) остались в платформе .NET 4.0 и, видимо, будут существовать еще некоторое время (возможно, длительное) в составе библиотеки базовых классов. Однако в рамках вышедшей версии .NET 3.0 компания Microsoft выпустила совершенно новый инструментальный набор для построения графических пользовательских интерфейсов под названием Windows Presentation Foundation (WPF). Как было показано в книге, WPF — довольно мощный инструмент, который позволяет создавать передовые пользовательские интерфейсы, и поэтому он стал рекомендуемым API-интерфейсом для построения современных графических пользовательских интерфейсов.

Это приложение задумано как ознакомительный тур по традиционному API-интерфейсу Windows Forms. Одна из причин заключается в том, что это полезно для понимания исходной модели программирования: в настоящее время существует много различных приложений Windows Forms, которые еще необходимо сопровождать. Кроме того, во многих графических пользовательских интерфейсах просто не нужны все возможности, предлагаемые WPF. Если понадобится создать более традиционное бизнес-приложение, которому не нужны все эти излишества, то часто вполне достаточно API-интерфейса Windows Forms.

В этом приложении вы ознакомитесь с моделью программирования Windows Forms, поработаете с интегрированными конструкторами Visual Studio, опробуете различные элементы управления Windows Forms и получите общее впечатление о программировании графики с помощью GDI+. Кроме того, вы соберете все это в единое целое, упаковав в (средней сложности) приложение рисования.

На заметку! Вот одно из доказательств того, что Windows Forms не собирается пропадать в ближайшем будущем: .NET 4.0 поставляется с совершенно новой сборкой Windows Forms под названием `System.Windows.Forms.DataVisualization.dll`. Эта библиотека позволяет добавлять в программы диаграммы с аннотациями, трехмерную графику и поддержку определения попадания. API-интерфейс Windows Forms из .NET 4.0 не рассматривается в данном приложении, но если вам понадобится информация о нем, просмотрите пространство 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 различные пространства имен сборки `System.Windows.Forms.dll`. На рис. А.1 показаны эти пространства имен, отображаемые в браузере объектов Visual Studio.

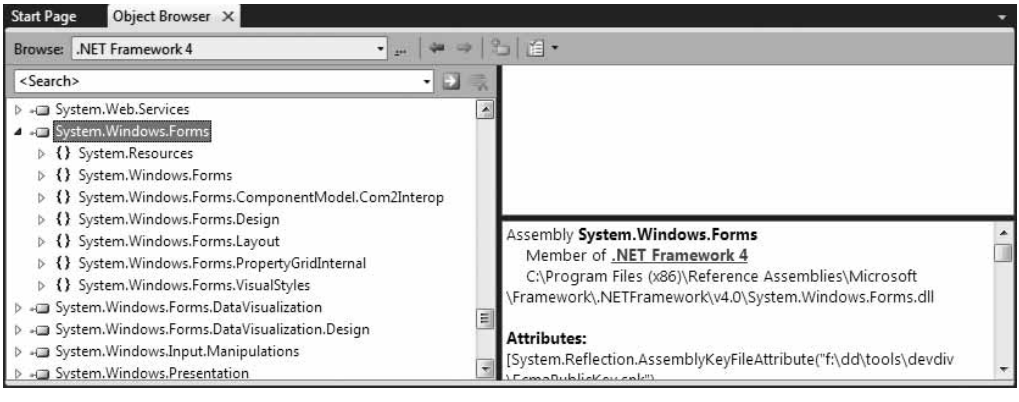

Рис. А.1. Пространства имен сборки `System.Windows.Forms.dll`

Несомненно, наиболее важным пространством имен Windows Forms является `System.Windows.Forms`. Типы из этого пространства имен можно разбить на следующие крупные категории.

- *Основная инфраструктура.* Это типы, представляющие основные операции программ Windows Forms (`Form` и `Application`), и различные типы, предназначенные для взаимодействия с унаследованными элементами ActiveX, а также для взаимодействия с новыми специальными элементами управления WPF.
- *Элементы управления.* Эти типы применяются для создания графических пользовательских интерфейсов (например, `Button`, `MenuStrip`, `ProgressBar` и `DataGridView`), и все они являются производными от базового класса `Control`. Элементы управления допускают настройку на этапе проектирования и видимы (по умолчанию) во время выполнения.
- *Компоненты.* Это типы, которые не порождены от базового класса `Control`, но все-таки могут предоставлять программам Windows Forms визуальные возможности (например, `ToolTip` и `ErrorProvider`). Многие компоненты (к примеру, `Timer` и `System.ComponentModel.BackgroundWorker`) не видимы во время выполнения, но все-таки допускают настройку на этапе проектирования.

- *Общие диалоговые окна.* В Windows Forms имеется несколько готовых диалоговых окон для выполнения распространенных операций (например, `OpenFileDialog`, `PrintDialog` и `ColorDialog`). Понятно, что если эти предлагаемые варианты чем-то не устраивают, на их основе можно построить собственные диалоговые окна.

Общее количество типов в пространстве имен `System.Windows.Forms` существенно больше 100, и было бы излишней тратой времени (и бумаги) перечислять все члены семейства Windows Forms. Однако по мере чтения данного приложения вы получите четкое понимание, которое позволит продвигаться дальше. В любом случае дополнительную информацию можно прочитать в документации .NET Framework 4.0 SDK.

Создание простого приложения Windows Forms

Конечно, современные IDE-среды .NET (Visual Studio, Visual C# Express и SharpDevelop) содержат многочисленные конструкторы форм, визуальные редакторы и интегрированные средства генерации кода (мастера), которые предназначены для облегчения создания приложений Windows Forms. Эти средства исключительно полезны, но они мешают процессу изучения Windows Forms, ведь они генерируют большой объем типового кода, который скрывает лежащую в основе объектную модель. Поэтому наш первый пример приложения Windows Forms будет проектом консольного приложения.

Вначале создайте консольное приложение по имени `SimpleWinFormsApp`. Затем выберите пункт меню Project Add Reference (Проект⇨Добавить ссылку) и на вкладке .NET открывшегося диалогового окна укажите сборки `System.Windows.Forms.dll` и `System.Drawing.dll`. Затем введите 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 Visual Studio обнаруживает класс, расширяющий `System.Windows.Forms.Form`, открывается соответствующий визуальный конструктор графического пользовательского интерфейса (если это первый класс в файле кода C#). Двойной щелчок на файле `Program.cs` приводит к открытию визуального конструктора, но пока не делайте это! Работа с визуальным конструктором Windows Forms рассматривается в следующем пример, а пока что щелкните правой кнопкой мыши на файле кода C# в Solution Explorer и в появившемся контекстном меню выберите пункт View Code (Просмотреть код).

Этот код представляет собой самое простое из всех возможных приложений Windows Forms. В качестве абсолютного минимума необходим класс, расширяющий базовый класс `Form`, и метод `Main()` для вызова статического метода `Application.Run()` (классы `Form` и `Application` будут рассмотрены далее в этом приложении). Если запустить полученное приложение, то поверх всех окон откроется окно, которое можно свернуть, развернуть, закрыть, а также изменить его размеры (рис. А.2).

Рис. А.2. Простое приложение Windows Forms

На заметку! Запустив эту программу, вы заметите под его верхним окном очертания окна командной строки. Причина в том, что по умолчанию компилятору C# передается флаг `/target:exe`. Чтобы не отображать окно командной строки, его можно изменить на `/target:winexe`. Для этого дважды щелкните на значке Properties (Свойства) в Solution Explorer и на вкладке Application (Приложение) в качестве Output Type (Тип вывода) укажите вариант Windows Application (Приложение Windows).

Действительно, полученное приложение не особенно впечатляет, но зато оно демонстрирует, насколько простым может быть приложение Windows Forms. Чтобы немного его улучшить, можно добавить в класс `MainWindow` специальный конструктор, который позволяет вызывающему коду устанавливать различные 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 изменить 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 (меню, строки состоянии, кнопки и т.д.), позволяющие вводить какую-то информацию. 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` (в цепочке наследования типа `Form`) определяет свойство `Controls`. Оно является оболочкой для специальной коллекции `ControlCollection`, вложенной в класс `Control`. Данная коллекция содержит ссылки на все элементы пользовательского интерфейса, поддерживаемые этим производным типом. Как и другие контейнеры, этот тип поддерживает несколько методов для вставки, удаления и поиска конкретного виждета пользовательского интерфейса (табл. А.1).

Таблица А.1. Члены `ControlCollection`

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

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

- определение переменной-члена нужного элемента пользовательского интерфейса в классе, производном от `Form`;
- конфигурирование внешнего вида и поведения элемента пользовательского интерфейса;
- добавление элемента пользовательского интерфейса в контейнер `ControlCollection` данной формы с помощью вызова `Controls.Add()`.

Предположим, что в класс `MainWindow` нужно добавить поддержку пункта меню File Exit (Файл⇨Выход). Ниже показаны необходимые изменения, а после них приводится анализ.

```
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 конкретный `ToolStripMenuItem` — любые пункты меню верхнего уровня (например, File) либо элементы подменю (такие как Exit), поддерживаемые содержащим их окном.

Система меню конфигурируется во вспомогательном методе `BuildMenuSystem()`. Текст каждого `ToolStripMenuItem` задается свойством `Text`; каждому пункту меню присваивается строковый литерал, содержащий встроенный символ амперсанда. Как известно, этот синтаксис позволяет использовать клавиатурные сокращения с клавишей <Alt>. Так, нажатие <Alt+F> активизирует меню File, а <Alt+X> — пункт меню Exit. Также обратите внимание, что объект `ToolStripMenuItem` (`mnuFile`) меню File добавляет подпункты с помощью свойства `DropDownItems`. Сам объект `MenuStrip` добавляет пункты меню верхнего уровня с помощью свойства `Items`.

После создания системы меню ее 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. Отображаемую систему меню можно менять на основе настроек пользователя или параметров безопасности.

Еще одним интересным моментом является обработка события `Click` пункта меню File Exit; она позволяет перехватывать момент выбора пользователем этого пункта. Событие `Click` работает в сочетании со стандартным типом делегата по имени `System.EventHandler`. Это событие может вызывать только методы, которые принимают `System.Object` в качестве первого параметра и `System.EventArgs` — в качестве второго. Здесь используется лямбда-выражение для завершения всего приложения с помощью статического метода `Application.Exit()`.

Перекомпилировав и запустив это приложение, вы увидите, что ваше простое окно уже содержит пользовательскую систему меню (рис. А.3).

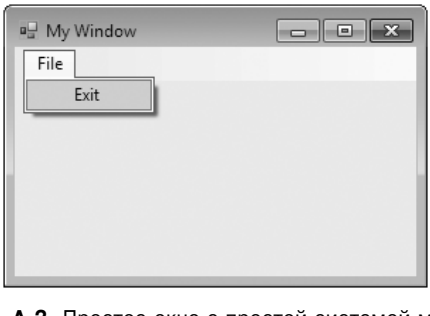

Рис. А.3. Простое окно с простой системой меню

Типы System.EventArgs и System.EventHandler

System.EventHandler — один из многих типов делегатов, применяемых 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, сгенерировавший данное событие. Предположим, к примеру, что нужно изменить реализацию лямбда-выражения следующим образом:

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

Можно проверить, что данное событие сгенерировано типом mnuFileExit, т.к. в окне сообщения отображается следующая строка:

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

А для чего нужен второй аргумент, т.е. System.EventArgs? В действительности мало для чего, поскольку он просто расширяет тип Object и практически не добавляет дополнительной функциональности:

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

Однако этот тип полезен в общей схеме обработки событий в .NET, т.к. он является предком многих производных типов. К примеру, тип MouseEventArgs расширяет тип EventArgs, добавляя информацию о текущем состоянии мыши. Тип KeyEventArgs, также расширяет EventArgs для сообщения информации о состоянии клавиатуры (например, какая клавиша была нажата); тип PaintEventArgs расширяет EventArgs для выдачи графических данных и т.д. Многочисленные потомки EventArgs (и использующие их делегаты) встречаются не только при работе с Windows Forms, но также и при работе с API-интерфейсами WPF и ASP.NET.

Вы можете и дальше добавлять различную функциональность в класс MainWindow (скажем, строку состояния и диалоговые окна) с помощью простого текстового редактора, но так недолго переутомиться — ведь придется вручную писать всю логику настройки элементов управления. К счастью, в Visual Studio встроены многочисленные визуальные конструкторы, которые могут позаботиться о таких деталях. При использовании этих средств в оставшейся части этого приложения не забывайте, что они просто генерируют обыденный код C#, и ничего магического в них нет.

Исходный код. Проект SimpleWinFormsApp находится в подкаталоге Appendix A.

Шаблон проекта 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) через 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).

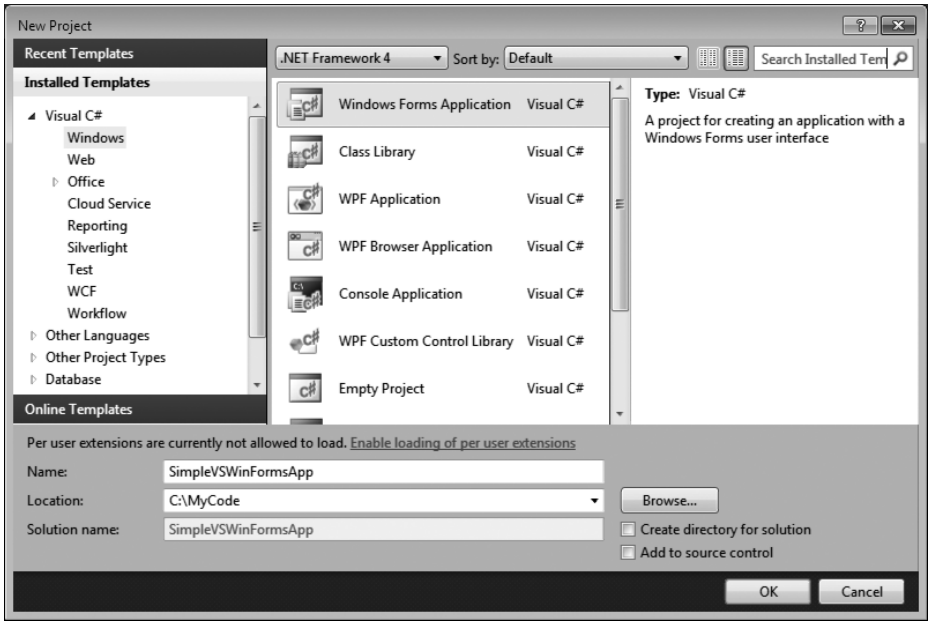

Рис. А.4. Шаблон проекта Windows Forms Application в Visual Studio

Поверхность визуального конструктора

Прежде чем приступить к созданию более интересных Windows-приложений, вначале воссоздадим предыдущий пример, но уже с помощью инструментов визуального проектирования. После создания нового проекта Windows Forms в Visual Studio появляется поверхность визуального конструктора, на которую можно перетаскивать любое количество элементов управления. Этот же конструктор позволяет настраивать начальный размер окна, просто изменяя размеры самой формы с помощью специальных скоб захвата (рис. А.5).

Рис. А.5. Визуальный конструктор форм

Если вы захотите сконфигурировать внешний вид окна (а также любого элемента в визуальном конструкторе форм), это можно сделать с помощью окна Properties (Свойства). Как и в проекте Windows Presentation Foundation, это окно позволяет при-

сваивать значения свойствам, а также устанавливать обработчики событий для элемента, выделенного в данный момент на поверхности конструктора (конфигурируемый элемент выбирается из раскрывающегося списка в верхней части окна Properties).

Пока форма не содержит ничего, и поэтому список состоит только из первоначальной формы, которой по умолчанию назначено имя `Form1` — это видно в свойстве `Name` (предназначенном только для чтения) на рис. А.6.

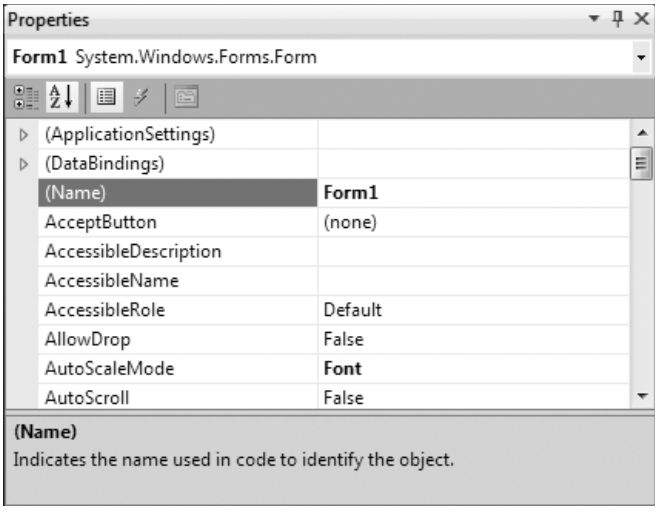

Рис. А.6. Окно Properties для установки свойств и обработчиков событий

На заметку! Окно Properties можно настроить на отображение содержимого по категориям или по алфавиту — для этого предназначены первые две кнопки прямо под раскрывающимся списком. Рекомендуется выбрать упорядочение по алфавиту, поскольку так легче быстро найти нужное свойство или событие.

Следующий элемент визуального конструктора, который следует знать — это окно проводника решений (Solution Explorer). Все проекты Visual Studio поддерживают это окно, но оно особенно полезно при создании приложений Windows Forms для быстрого изменения имени файла и соответствующего класса для любого окна и просмотра файла, который содержит сопровождаемый конструктором код (об этом интересном моменте будет рассказано немного позже). Пока что щелкните правой кнопкой мыши на значке `Form1.cs` и выберите в контекстном меню пункт Rename (Переименовать). Назовите первоначальное окно более осмысленно: `MainWindow.cs`. Среда IDE спросит, хотите ли вы изменить имя начального класса — согласитесь на это.

Разбор первоначальной формы

Перед построением системы меню необходимо точно разобраться в том, что 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 (Просмотр кода). Обратите внимание, что форма определена как частичный тип, что позволяет определить один тип 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 соответствующего файла кода C#, содержащего производный от `Form` тип. Используя окна Solution Explorer, откройте файл `MainWindow.Designer.cs`. Теперь взгляните на следующий код (для простоты из него убраны комментарии; ваш код может слегка отличаться, в зависимости от настроек, выполненных в окне Properties):

```
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 `IContainer` и метод `Dispose()` — это всего лишь инфраструктура, используемая инструментами визуального проектирования Visual Studio. Однако обратите внимание, что метод `InitializeComponent()` активно используется, причем не только конструктором формы во время выполнения: среда Visual Studio применяет этот метод во время проектирования для корректной визуализации пользовательского интерфейса на поверхности визуального конструктора форм. Чтобы удостовериться в этом, измените значение, присваиваемое свойству `Text` текущей формы, на `"My Main Window"`. После активизации визуального конструктора заголовок формы изменится соответствующим образом.

При использовании инструментов визуального проектирования (т.е. 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%); при следующей компиляции и запуске 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 помощью Visual Studio. Если вы вставите в InitializeComponent() синтаксически (или логически) ошибочный 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, IDE-среда может просто удалить его! Всегда помните, что каждое окно приложения Windows Forms составлено с использованием частичных классов.

Разбор класса Program

Кроме кода реализации 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 на типе Application, чтобы установить некоторые базовые параметры визуализации. Обратите внимание, что метод Main() снабжен атрибутом [STAThread]: это 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, сопровождаемую COM — *однопоточный апартамент*. В сущности, это гарантирует, что объекты COM будут безопасными к потокам, даже если автор конкретного COM-объекта не предусмотрел для этого случая специальный код.

Визуальное построение системы меню

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

Перетащите элемент MenuStrip в верхнюю часть визуального конструктора форм. Среда Visual Studio отреагирует открытием редактора меню. Если внимательно присмотреться к этому редактору, можно увидеть в верхнем правом углу элемента маленький треугольник.

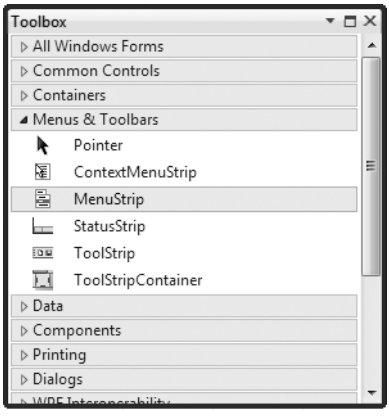

Рис. А.7. Элементы управления Windows Forms, которые можно добавлять на поверхность визуального конструктора

Щелчок на нем открывает контекстно-чувствительный встроенный редактор, который позволяет установить значения сразу нескольких свойств (аналогичные 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.

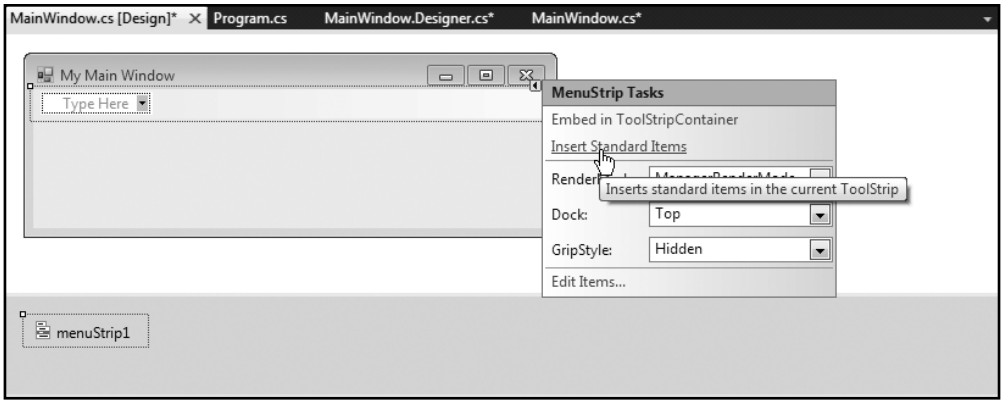

Рис. А.8. Встроенный редактор меню

В этом примере среда Visual Studio создала за вас всю систему меню. А теперь откройте сопровождаемый конструктором файл (`MainWindow.Designer.cs`) — обратите внимание, что в методе `InitializeComponent()` появилось несколько новых переменных-членов, которые представляют созданную систему меню. И, наконец, вернитесь в окно визуального конструктора и отмените последнюю операцию, нажав на клавиатуре комбинацию <Ctrl+Z>. После этого вы опять будете находиться в редакторе меню, а сгенерированный код исчезнет. С помощью конструктора меню введите самый верхний пункт меню File (Файл) и его подпункт Exit (Выход), как показано на рис. А.9.

Если вы просмотрите код метода `InitializeComponent()`, то увидите там примерно такой же код, который вы вводили вручную в первом примере данного приложения. В заключение упражнения вернитесь в окно визуального конструктора форм и щелкните на кнопке со значком молнии в окне Properties. Будут показаны все события, на которые можно реагировать для выбранного элемента.

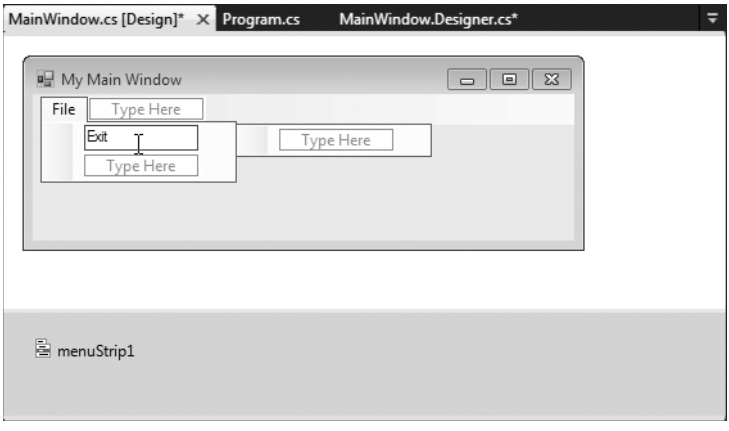

Рис. А.9. Ручное построение системы меню

Для выбранного меню Exit (имеющего стандартное имя `exitToolStripMenuItem`) найдите событие `Click` (рис. А.10).

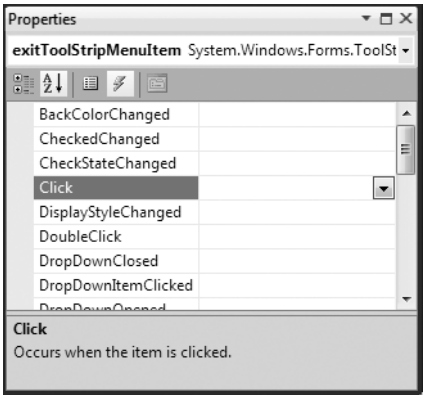

Рис. А.10. Обработка событий с помощью IDE-среды

Теперь можно ввести имя метода, который будет вызываться при щелчке на элементе, или же просто дважды щелкнуть на событии из списка в окне Properties. В последнем случае IDE-среда выберет за вас имя обработчика события (по образцу *ИмяЭлемента_ИмяСобытия*()). При любом варианте 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();
  }
}
```

Загляните в код `InitializeComponent()` — в нем также учтены все последние изменения:

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

К этому моменту вы должны более свободно оперировать IDE-средой при создании приложений Windows Forms. Конечно, существует множество дополнительных клавиатурных сочетаний, редакторов и интегрированных мастеров генерации кода, но изложенной информации вполне достаточно для продолжения работы.

Внутреннее устройство формы

Итак, вы уже умеете создавать простые приложения Windows Forms с помощью Visual Studio (или без нее). Теперь пора хорошо разобраться в типе `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` содержит много функциональности, унаследованной от родительских классов, а также из реализуемых им многочисленных интерфейсов.

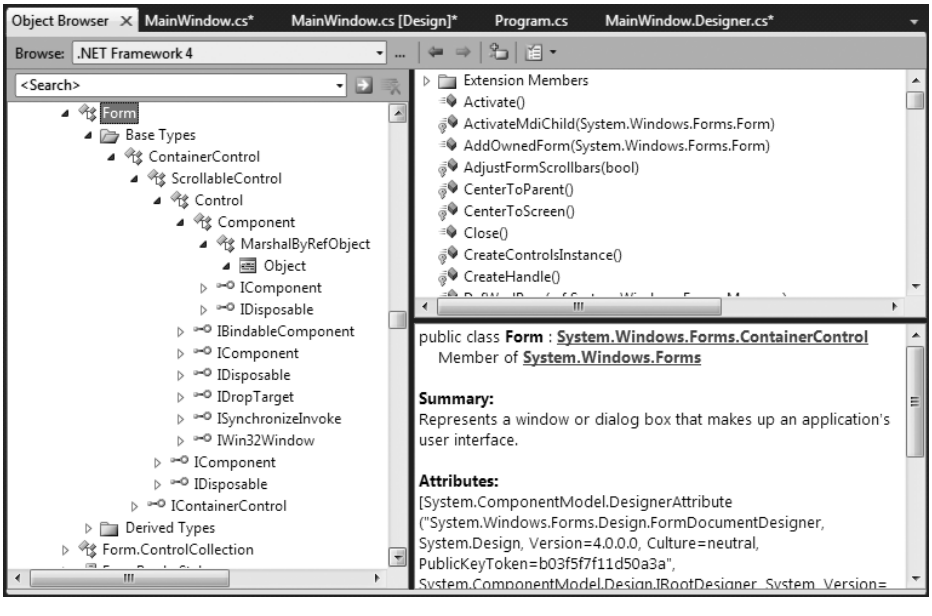

Рис. А.11. Цепочка наследования для типа `System.Windows.Forms.Form`

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

Хотя при порождении типа `Form` используются многие другие базовые классы и интерфейсы, даже профессиональному разработчику Windows Forms совсем *не* обязательно знать роли всех членов всех классов или реализованных интерфейсов. В действительности большинство членов (в частности, свойств и событий) можно легко устанавливать с помощью окна Properties в Visual Studio. Однако важно понимать функциональность, предоставляемую родительскими классами `Control` и `Form`.

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

Родительский класс	Описание
`System.Object`	Как и любой тип в .NET, класс `Form` является `object`
`System.MarshalByRefObject`	К типам, производным от этого класса, можно удаленно обратиться 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, хотя он не обязательно является видимым во время выполнения
`System.Windows.Forms.Control`	Определяет общие члены пользовательского интерфейса для всех элементов управления Windows Forms, включая и сам тип `Form`
`System.Windows.Forms.ScrollableControl`	Определяет поддержку горизонтальных и вертикальных полос прокрутки, а также членов, управляющих окном просмотра в прокручиваемой области
`System.Windows.Forms.ContainerControl`	Содержит функции управления фокусом ввода для элементов, которые могут служить в качестве контейнеров для других элементов управления
`System.Windows.Forms.Form`	Представляет любую специальную форму, дочернюю форму MDI или диалоговое окно

Функциональность класса `Control`

Класс `System.Windows.Forms.Control` задает общее поведение, необходимое для любого типа графического пользовательского интерфейса. Основные члены класса `Control` позволяют настраивать размер и позицию элемента, захватывать данные клавиатуры и мыши, получать или задавать фокус/видимость члена и т.д. В табл. А.3 приведены наиболее интересные свойства, которые сгруппированы по схожести функций.

Таблица А.3. Основные свойства типа `Control`

Свойство	Описание

`BackColor`
`ForeColor`
`BackgroundImage`
`Font`
`Cursor`
	Эти свойства определяют основные характеристики элемента управления (цвет, шрифт 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

Окончание табл. А.3

Свойство	Описание

`Enabled`
`Focused`
`Visible`
	Эти свойства содержат логические значения, которые задают состояние текущего элемента управления

`ModifierKeys`	Это статическое свойство проверяет текущее состояние клавиш-модификаторов (<Shift>, <Ctrl> и <Alt>) и возвращает это состояние в типе `Keys`
`MouseButtons`	Это статическое свойство проверяет текущее состояние кнопок мыши (левая, правая и средняя) и возвращает это состояние в типе `MouseButtons`
`TabIndex` `TabStop`	Эти свойства позволяют настроить 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, связанные с элементом управления
`Controls`	Это свойство предоставляет доступ к строго типизированной коллекции (например, `ControlsCollection`), которая содержит все дочерние элементы управления данного элемента управления

Как и можно было предположить, класс `Control` определяет ряд событий, которые позволяют перехватывать действия, связанные с мышью, клавиатурой, отображением и перетаскиванием. Некоторые полезные события, сгруппированные по функциональности, приведены в табл. А.4.

Таблица А.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 службой графической визуализации GDI+

И, наконец, базовый класс `Control` определяет несколько методов, которые позволяют взаимодействовать с любым типом, производным от `Control`.

Просматривая методы типа Control, обратите внимание, что имена многих из них содержат префикс On, за которым следует имя конкретного события (например, OnMouseMove, OnKeyUp и OnPaint). Каждый из таких виртуальных методов с префиксом On представляет собой стандартный обработчик для соответствующего события. Если вы переопределите любой из этих виртуальных методов, то получите возможность выполнять всю необходимую пред- и постобработку события перед и после вызова стандартной реализации в родительском классе:

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

В некоторых случаях это может быть удобно (особенно при создании специального элемента управления, производного от стандартного элемента), но события часто обрабатываются с помощью стандартного синтаксиса работы с событиями в C# (на самом деле это стандартное поведение визуальных конструкторов Visual Studio). При такой обработке событий инфраструктура вызывает специальный обработчик события после завершения работы родительской реализации. Например, следующий код позволяет вручную обрабатывать 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 описанным методам On*XXX*() следует знать еще несколько 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 далее в этом приложении).

Функциональность класса Form

Класс Form обычно (хотя и не обязательно) является непосредственным базовым классом для специальных типов Form. Кроме обширного набора членов, унаследованных от классов Control, ScrollableControl и ContainerControl, тип Form добавляет дополнительные функциональные возможности — в особенности для главных окон, до-

черних окон MDI и диалоговых окон. Сначала ознакомимся с основными свойствами, приведенными в табл. А.5.

Таблица А.5. Свойства типа `Form`

Свойство	Описание
`AcceptButton`	Получает или устанавливает кнопку на форме, для которой имитируется щелчок при нажатии клавиши <Enter>

`ActiveMdiChild`
`IsMdiChild`
`IsMdiContainer`
	Используются в контексте MDI-приложений

`CancelButton`	Получает или устанавливает кнопочный элементу правления на форме, для которой имитируется щелчок при нажатии клавиши <Esc>
`ControlBox`	Получает или устанавливает значение, которое указывает, имеется ли на форме область управления (т.е. значки, позволяющие свернуть, развернуть и закрыть форму в правом верхнем углу окна)
`FormBorderStyle`	Получает или устанавливает стиль рамки формы. Используется в сочетании с перечислением `FormBorderStyle`
`Menu`	Получает или устанавливает меню, расположенное на форме
`MaximizeBox` `MinimizeBox`	Определяют, включены ли на форме значки развертывания и свертывания
`ShowInTaskbar`	Определяет, будет ли отображаться форма в панели задач Windows
`StartPosition`	Получает или устанавливает первоначальную позицию формы во время выполнения, заданную с помощью перечисления `FormStartPosition`
`WindowState`	Конфигурирует отображение формы при запуске. Используется в сочетании с перечислением `FormWindowState`

Кроме многочисленных стандартных обработчиков событий с префиксом `On`, тип `Form` определяет ряд основных методов, перечисленных в табл. А.6.

Таблица А.6. Основные методы типа `Form`

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

И, наконец, в классе `Form` определен набор событий, большинство из которых генерируются во время жизненного цикла формы (табл. А.7).

Таблица А.7. События типа `Form`

Событие	Описание
`Activated`	Это событие возникает при *активизации* формы, т.е. при получении фокуса на рабочем столе
`FormClosed` `FormClosing`	Эти события возникают непосредственно перед закрытием формы и сразу после закрытия
`Deactivate`	Это событие возникает при *деактивизации* формы, т.е. при потере фокуса на рабочем столе
`Load`	Это событие возникает после размещения формы в памяти, но еще до отображения на экране
`MdiChildActive`	Это событие возникает при активизации дочернего окна

Жизненный цикл типа `Form`

Если вам приходилось программировать пользовательские интерфейсы с помощью инструментальных наборов для построения графических пользовательских интерфейсов наподобие Java Swing, Mac OS X Cocoa или WPF, то вы знаете, что *оконные типы* поддерживают множество событий, генерируемых во время их жизненного цикла. Это верно и для Windows Forms. Как вы уже знаете, жизненный цикл формы начинается тогда, когда вызывается конструктор класса перед его передачей методу `Application.Run()`.

Когда объект помещен в управляемую кучу, инфраструктура генерирует событие `Load`. В обработчике события `Load` можно сконфигурировать внешний вид объекта `Form`, подготовить содержащиеся в нем дочерние элементы управления (такие как `ListBox` и `TreeView`) или распределить ресурсы, необходимые для работы формы (например, подключения к базам данных и прокси для удаленных объектов).

После события `Load` генерируется событие `Activated`, но только тогда, когда форма получает фокус как активное окно на рабочем столе. Логической противоположностью событию `Activated` является событие `Deactivate`, которое возникает, когда форма перестает быть активным окном. Понятно, что события `Activated` и `Deactivate` могут возникать неоднократно за время жизни конкретного объекта `Form`, по мере того, как пользователь переключается между активными окнами и приложениями.

Если пользователь решает закрыть данную форму, возникают два события: `FormClosing` и `FormClosed`. Событие `FormClosing` генерируется первым и удобно для вывода конечному пользователю надоедливого (но полезного) сообщения вроде “Вы *уверены*, что хотите закрыть данное приложение?” Этот шаг позволяет пользователю сохранить все нужные данные, прежде чем закрыть приложение.

Событие `FormClosing` работает в сочетании с делегатом `FormClosingEventHandler`. Если установить свойство `FormClosingEventArgs.Cancel` в `true`, то окно не будет уничтожено и просто возвратится к нормальной работе. А если `FormClosingEventArgs.Cancel` равно `false`, то возникает событие `FormClosed`, и приложение Windows Forms завершает работу, выгружает домен приложения и завершает процесс.

Приведенный ниже фрагмент кода изменяет конструктор формы и обрабатывает события `Load`, `Activated`, `Deactivate`, `FormClosing` и `FormClosed` (вспомните, что IDE-среда автоматически генерирует нужный делегат и обработчик события при двукратном нажатии клавиши <Tab> после ввода символов +=):

```
public MainWindow()
{
  InitializeComponent();

  // Обработать различные 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);
}
```

В обработчиках событий `Load`, `FormClosed`, `Activated` и `Deactivate` нужно изменить значение новой строковой переменной-члена уровня `Form` (по имени `lifeTimeInfo`) — это просто сообщение, которое выводит имя перехваченного события. Сначала добавьте этот член в свой класс, производный от `Form`:

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

После этого необходимо реализовать обработчики событий. Значение строки `lifeTimeInfo` выводится в диалоговом окне в обработчике события `FormClosed`:

```
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` выводится запрос, действительно ли пользователь желает завершить работу приложения с входными аргументами `FormClosingEventArgs`. В приведенном ниже коде метод `MessageBox.Show()` возвращает объект `DialogResult`, содержащий значение, которое представляет кнопку, выбранную конечным пользователем. В нашем случае диалоговое окно содержит кнопки Yes (Да) и No (Нет); поэтому значение, возвращаемое методом `Show()`, 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 щелкнул пользователь?
  if (dr == DialogResult.No)
    e.Cancel = true;
  else
    e.Cancel = false;
}
```

Наконец, давайте внесем последнее изменение. В настоящий момент пункт меню File Exit (Файл⇨Выход) разрушает все приложение, что нежелательно. Чаще всего обработчик File Exit окна верхнего уровня вызывает унаследованный метод `Close()`, который генерирует события, связанные с закрытием, и лишь затем уничтожает приложение:

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

Теперь запустите полученное приложение и несколько раз переместите фокус на форму и вне ее (для запуска событий `Activated` и `Deactivate`). При закрытии приложения появится диалоговое окно, которое выглядит примерно так, как показано на рис. A.12.

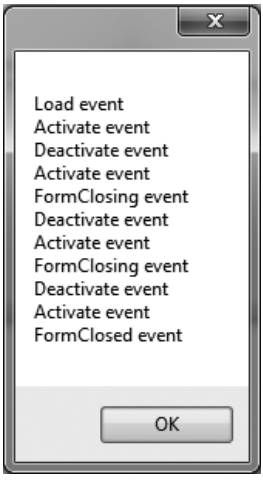

Рис. A.12. Жизненный цикл и отдельные моменты производного от `Form` типа

Исходный код. Проект `SimpleVSWinFormsApp` находится в подкаталоге `Appendix A`.

Реагирование на действия мыши и клавиатуры

Возможно, вы помните, что родительский класс `Control` определяет набор событий, которые позволяют различными способами отслеживать действия мыши и клавиатуры. Чтобы удостовериться в этом, создайте новый проект приложения Windows Forms с именем `MouseAndKeyboardEventsApp`, измените (используя Solution Explorer) первоначальное имя формы на `MainWindow.cs` и обработайте событие `MouseMove` с помощью окна Properties. В результате будет сгенерирован следующий обработчик события:

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

Событие `MouseMove` работает в сочетании с делегатом `System.Windows.Forms.MouseEventHandler`. Этот делегат может вызывать только методы, у которых первый параметр имеет тип `System.Object`, а второй — тип `MouseEventArgs`. Данный 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 являются самоочевидными, но в табл. А.8 приведены более конкретные детали.

Таблица А.8. Свойства типа `MouseEventArgs`

Свойство	Описание
`Button`	Позволяет получить кнопку мыши, на которой был выполнен щелчок (см. перечисление `MouseButtons`)
`Clicks`	Позволяет получить количество нажатий и отпусканий кнопки мыши
`Delta`	Позволяет получить количество делений (которые соответствуют одному сдвигу колесика мыши) со знаком для текущего поворота колесика мыши
`Location`	Позволяет получить объект `Point`, содержащий координаты *x* и *y* мыши
`X`	Позволяет получить координату *x* щелчка мышью
`Y`	Позволяет получить координату *y* щелчка мышью

Теперь можно 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);
}
```

Если запустить это приложение и перемещать курсор мыши по окну, то позиция курсора будет отображаться в поле заголовка типа `MainWindow` (рис. А.13).

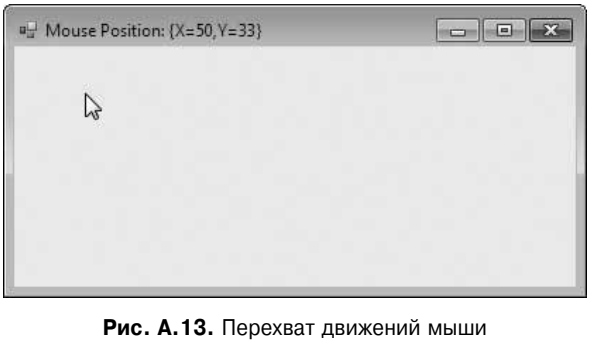

Рис. А.13. Перехват движений мыши

Определение кнопки мыши, которой был выполнен щелчок

Еще одно действие, которое обычно требуется выполнять при работе с мышью — определение, какой кнопкой был выполнен щелчок при возникновении события MouseUp, MouseDown, MouseClick или MouseDoubleClick. Когда нужно точно знать кнопку мыши (левую, правую или среднюю), необходимо проверить значение свойства Button класса MouseEventArgs. Значения свойства Button берутся из соответствующего перечисления 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 проверить в действии за счет обработки события MouseDown типа MainWindow с помощью окна Properties. Приведенный ниже обработчик события MouseDown выводит кнопку мыши, щелчок которой был выполнен в пределах окна с сообщением:

```
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 Windows-приложениях обычно имеется множество элементов управления вводом данных (например, TextBox), в которых пользователь может вводить информацию с помощью клавиатуры. При таком перехвате клавиатурного ввода нет необходимости в явной обработке событий клавиатуры, т.к. текстовую информацию можно извлечь из элемента управления с помощью различных свойств (например, свойства Text в случае TextBox).

Но если вам понадобится отслеживать ввод с клавиатуры для более экзотических целей (например, для фильтрации символов, поступающих в элемент управления или захвата нажатий клавиш на самой форме), то для таких случаев в библиотеках базовых классов предусмотрены события KeyUp и KeyDown. Эти события работают в сочетании с делегатом KeyEventHandler, который может указывать на любой метод, принимающий object в качестве первого параметра и тип KeyEventArgs в качестве второго. Этот тип определяется так:

```
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 приведены некоторые наиболее интересные свойства, поддерживаемые типом `KeyEventArgs`.

Таблица А.9. Свойства типа `KeyEventArgs`

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

Все это можно увидеть в действии, обработав событие `KeyDown`:

```
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 полученную программу. Теперь вы можете узнать не только, какой кнопкой мыши был выполнен щелчок, но и какая клавиша была нажата. Например, на рис. А.14 показан результат одновременного нажатия клавиш <Ctrl> и <Shift>.

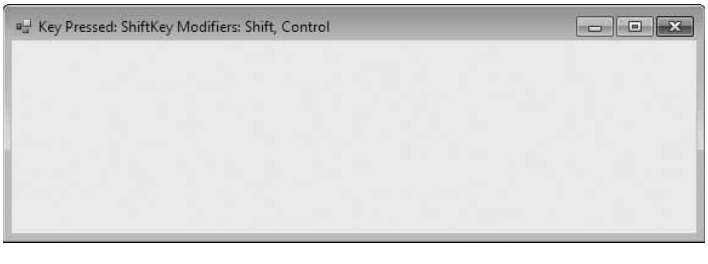

Рис. А.14. Перехват действий клавиатуры

Исходный код. Проект `MouseAndKeyboardEventsApp` находится в подкаталоге `Appendix A`.

Проектирование диалоговых окон

В программах с графическим пользовательским интерфейсом диалоговые окна являются одним из основных способов ввода пользовательской информации для применения в самом приложении. В отличие от других API-интерфейсов, с которыми вы, возможно, имели дело, в Windows Forms отсутствует базовый класс `Dialog`. Все диалоговые окна являются просто типами, производными от класса `Form`.

Как правило, диалоговые окна не должны менять свой размер, поэтому свойству `FormBorderStyle` присваивается значение `FormBorderStyle.FixedDialog`. Кроме того, свойства `MinimizeBox` и `MaximizeBox` обычно устанавливаются в `false`. В этом случае диалоговое окно имеет постоянный размер. А если установить в `false` свойство `ShowInTaskbar`, то форма не будет отображаться на панели задач Windows.

Давайте посмотрим, как проектировать диалоговые окна и работать с ними. Создайте новый проект Windows Forms Application под названием `CarOrderApp` и измените с помощью Solution Explorer первоначальное имя файла `Form1.cs` на `MainWindow.cs`. Теперь воспользуйтесь визуальным конструктором форм, чтобы создать простое меню File Exit (Файл⇨Выход), а также пункт меню Tool Order Automobile... (Сервис⇨Заказ автомобиля...). Вспомните, что для создания меню необходимо перетащить элемент `MenuStrip` из панели инструментов, а затем в окне конструктора настроить пункты меню. После этого понадобится реализовать обработчики события `Click` для пунктов меню Exit и Order Automobile с помощью окна Properties.

Обработчик пункта меню File Exit завершает работу приложения вызовом `Close()`:

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

Теперь в меню Project (Проект) Visual Studio выберите пункт Add Windows Forms (Добавить форму Windows Forms) и назовите новую форму `OrderAutoDialog.cs` (рис. А.15).

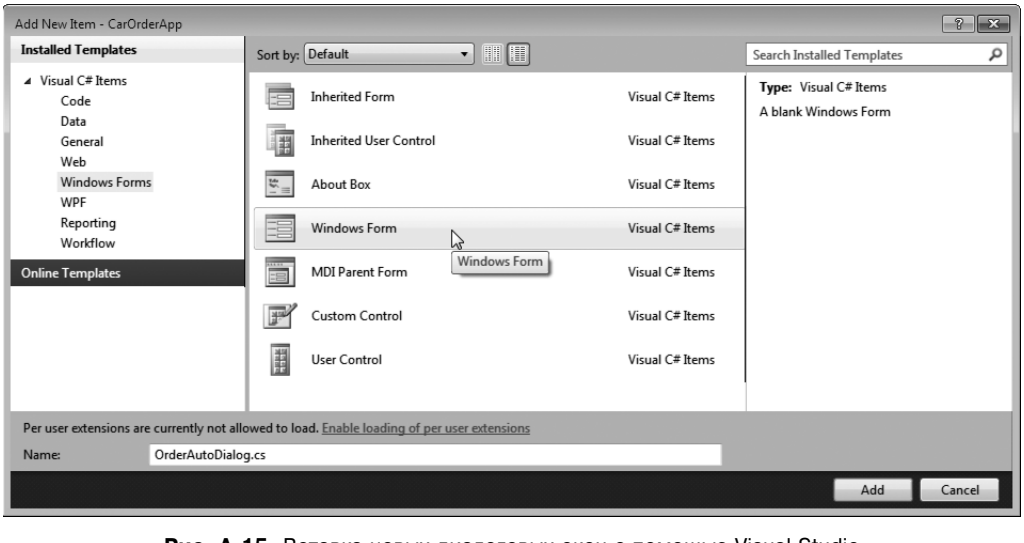

Рис. А.15. Вставка новых диалоговых окон с помощью Visual Studio

В рассматриваемом примере создайте диалоговое окно с традиционными кнопками OK и Cancel (которые называются соответственно `btnOK` и `btnCancel`), а также тремя полями `TextBox` с именами `txtMake`, `txtColor` и `txtPrice`. 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`

И, наконец, выберите кнопку OK и с помощью окна Properties установите свойство `DialogResult` в `OK`. Аналогично установите свойство `DialogResult` кнопки Cancel в `Cancel`. Как вы вскоре убедитесь, свойство `DialogResult` действительно полезно, поскольку 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. А.16 показан пример проектирования диалогового окна; для наглядности в нем даже добавлено несколько меток.

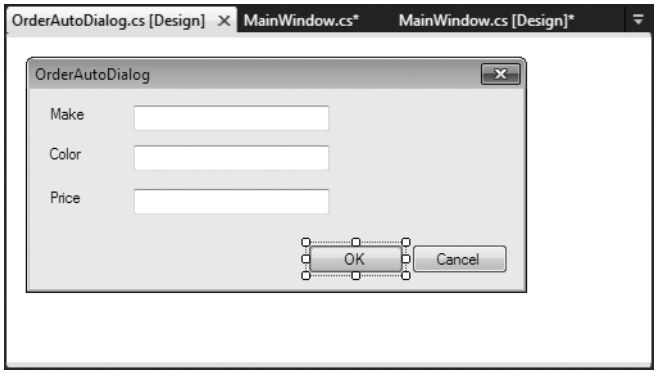

Рис. А.16. Тип `OrderAutoDialog`

Настройка порядка обхода по клавише <Tab>

Итак, диалоговое окно уже выглядит довольно привлекательно. Следующий шаг заключается в формализации порядка обхода при нажатии клавиши <Tab>. Вы уже знаете, что многие пользователи привыкли перемещать фокус ввода с помощью клавиши <Tab>, если форма содержит несколько графических элементов. Настройка последовательности переходов требует знакомства с двумя свойствами: `TabStop` и `TabIndex`.

Свойство `TabStop` может принимать значения `true` или `false`, в зависимости от того, хотите ли вы, чтобы данный элемент получал фокус с помощью клавиши <Tab>. Если свойство `TabStop` для какого-то элемента равно `true`, то для этого элемента можно установить свойство `TabOrder`, чтобы задать порядок активизации в последовательности переходов (нумерация с нуля), например:

```
// Настройка свойств перехода при нажатии <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 имеется мастер порядка обхода (Tab Order Wizard), который вызывается через пункт меню View Tab Order (Вид⇨Порядок обхода). Учтите, что этот пункт доступен, только если активен визуальный конструктор форм. После активизации для каждого графического элемента формы выводится текущее значение `TabIndex`. Чтобы изменить эти значения, щелкните на каждом элементе в той очередности, в которой вы хотите выполнять обход по нажатию <Tab> (рис. А.17).

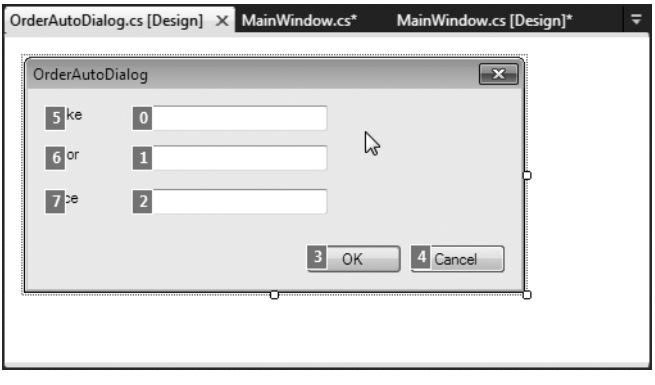

Рис. А.17. Мастер порядка обхода

Завершить мастер порядка обхода можно, нажав клавишу <Esc>.

Установка стандартной кнопки ввода для формы

Во многих формах, требующих от пользователя ввода каких-то данных (особенно в диалоговых окнах) имеется специальная кнопка, которая автоматически реагирует на нажатие пользователем клавиши <Enter>. Предположим, что при нажатии пользователем клавиши <Enter> необходимо вызвать обработчик события `Click` для кнопки `btnOK`. Для этого достаточно установить свойство `AcceptButton` следующим образом (это же можно сделать и с помощью окна Properties):

```
public partial class OrderAutoDialog : Form
{
  public OrderAutoDialog()
  {
    InitializeComponent();

    // Реакция на нажатие клавиши <Enter> такая же,
    // как если бы был совершен щелчок на кнопке btnOK.
    this.AcceptButton = btnOK;
  }
}
```

На заметку! Некоторые формы требуют возможности имитации щелчка на кнопке Cancel при нажатии пользователем клавиши <Esc>. Это можно сделать, присвоив свойству `CancelButton` формы объект `Button`, который соответствует кнопке Cancel.

Отображение диалоговых окон

При отображении диалогового окна первым делом потребуется 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 данное диалоговое окно. Примером модального окна может служить большинство окон "О программе". Чтобы открыть новое диалоговое окно, нужно вызвать метод `ShowDialog()` на объекте этого диалогового окна. Для отображения немодального диалогового окна предназначен метод `Show()`, который позволяет пользователю переключаться между диалоговым и главным окном (подобно окну "Найти/заменить").

В текущем примере мы изменим обработчик пункта меню Tools Order Automobile... для типа `MainWindow`, чтобы объект `OrderAutoDialog` выводился в модальном режиме. Вот первоначальный код:

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

  // Отобразить в виде модального диалогового окна и выяснить, на какой кнопке
  // был выполнен щелчок, с помощью возвращаемого значения 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, представляющего владельца диалогового окна (для формы, загружающей диалоговое окно, это `this`). Указание владельца диалогового окна устанавливает z-упорядочение форм, а заодно гарантирует (в случае немодального диалогового окна), что при закрытии главного окна будут закрыты и все принадлежащие ему окна.

Не забывайте, что при создании экземпляра типа, производного от `Form` (в данном случае `OrderAutoDialog`), окно *не* отображается на экране, а лишь размещается в памяти. А видимым оно становится лишь после вызова `Show()` или `ShowDialog()`. Кроме того, учтите, что метод `ShowDialog()` возвращает значение `DialogResult`, которые было присвоено одной из кнопок (метод `Show()` просто возвращает `void`). После возврата из метода `ShowDialog()` форма не отображается на экране, но по-прежнему находится в памяти. Это означает, что можно извлечь значения из каждого элемента `TextBox`. Однако при компиляции следующего кода будут выданы сообщения об ошибках:

```
private void orderAutomobileToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Создать объект диалогового окна.
  OrderAutoDialog dlg = new OrderAutoDialog();
  // Отобразить в виде модального диалогового окна и выяснить, на какой кнопке
  // был выполнен щелчок, с помощью возвращаемого значения 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!");
  }
}
```

Ошибки компиляции возникают из-за того, что Visual Studio объявляет элементы, добавляемые на поверхность визуального конструктора форм, как *закрытые* переменные-члены класса. Можете удостовериться в этом сами, открыв файл `OrderAutoDialog.Designer.cs`.

Хотя, строго говоря, диалоговое окно могло бы сохранить инкапсуляцию за счет добавления открытых свойств для установки и получения значений в текстовых полях, можно поступить проще: переопределить их с применением ключевого слова `public`. Для этого выберите в конструкторе каждый элемент `TextBox` и установите его свойство `Modifiers` в `Public` (с помощью окна Properties). После этого лежащий в основе код визуального конструктора будет выглядеть так:

```
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 можно скомпилировать и запустить приложение. После отображения диалогового окна и щелчка на кнопке OK вы увидите входные данные в окне сообщения.

Наследование форм

До сих пор все специальные и диалоговые окна, рассмотренные в этом приложении, были порождены непосредственно от класса `System.Windows.Forms.Form`. Однако одним из интересных аспектов Windows Forms является то, что типы `Form` могут служить базовыми классами для производных типов `Form`. Предположим, к примеру, что вы создали библиотеку кода .NET, которая содержит все основные диалоговые окна компании. Но потом вы решили, что в окно "О программе" неплохо было бы добавить трехмерный логотип компании. В этом случае можно не пересоздавать заново все окно, а расширить базовое окно "О программе", унаследовав прежний внешний вид:

```
// 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). Однако на этот раз выберите вариант Inherited Form (Унаследованная форма) и назовите новую форму `ImageOrderAutoDialog.cs` (рис. А.18).

При выборе этого варианта открывается диалоговое окно Inheritance Picker (Выбор наследования), в котором выводятся все формы из текущего проекта. Кнопка Browse (Обзор) позволяет выбрать форму из внешней сборки .NET. В рассматриваемом примере просто выберите класс `OrderAutoDialog`.

На заметку! Чтобы формы проекта отображались в диалоговом окне Inheritance Picker, необходимо хотя бы раз скомпилировать проект, т.к. для отображения вариантов это средство использует метаданные сборки.

После щелчка на кнопке OK инструменты визуального проектирования выводят все базовые элементы управления на их родительских элементах, и у каждого элемента слева вверху имеется значок со стрелкой, символизирующий наследование.

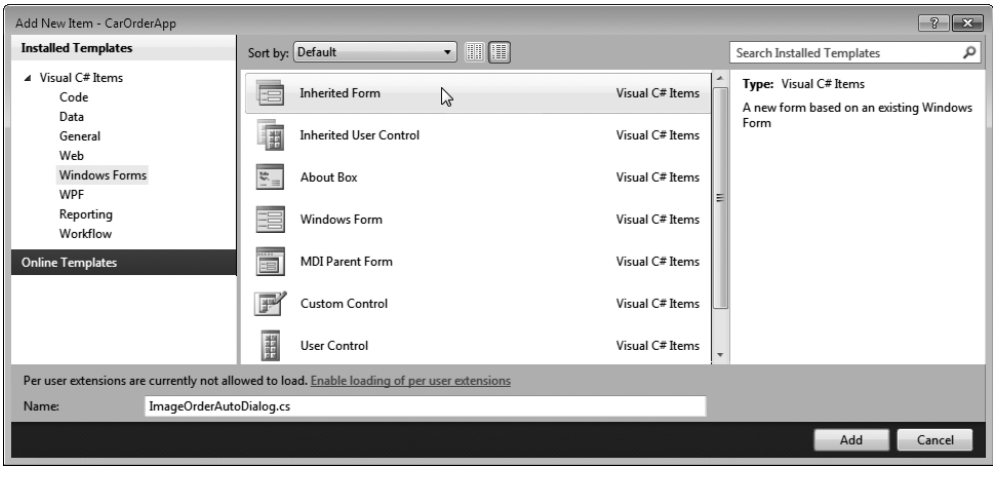

Рис. А.18. Добавление в проект производной формы

Чтобы завершить построение производного диалогового окна, найдите элемент `PictureBox` в разделе Common Controls (Общие элементы) панели инструментов и добавьте его на производную форму. Затем с помощью свойства `Image` выберите нужный файл с изображением. На рис. А.19 показан один из возможных пользовательских интерфейсов.

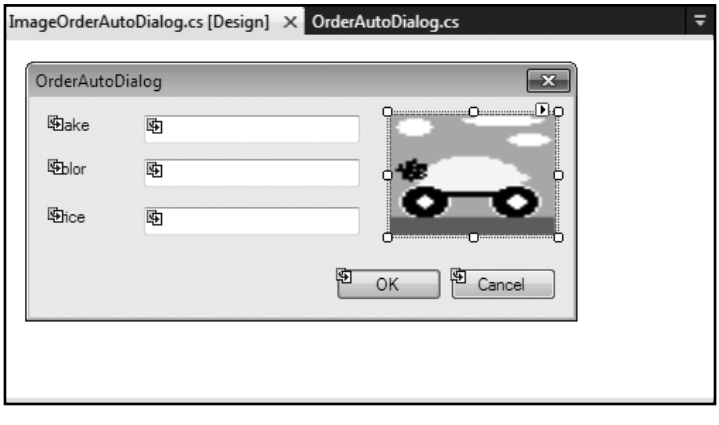

Рис. А.19. Пользовательский интерфейс класса `ImageOrderAutoDialog`

Теперь можно изменить обработчик события `Click` для пункта меню Tools Order Automobile..., чтобы создать экземпляр производного типа, а не базового класса `OrderAutoDialog`:

```
private void orderAutomobileToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Создать объект производного объекта 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 могут динамически генерировать графические данные, которые затем отображаются на поверхности окна. Предположим, например, что из реляционной базы данных выбран набор записей, и на их основе нужно вывести круговую (или столбчатую) диаграмму, которая наглядно показывает наличие товаров на складе. Или, может быть, вы захотите создать старую видеоигру, но уже на платформе .NET. Независимо от цели, API-интерфейс под названием GDI+ позволяет визуализировать графические данные в приложениях Windows Forms. Эта технология сконцентрирована в сборке `System.Drawing.dll`, которая определяет несколько пространств имен (рис. А.20).

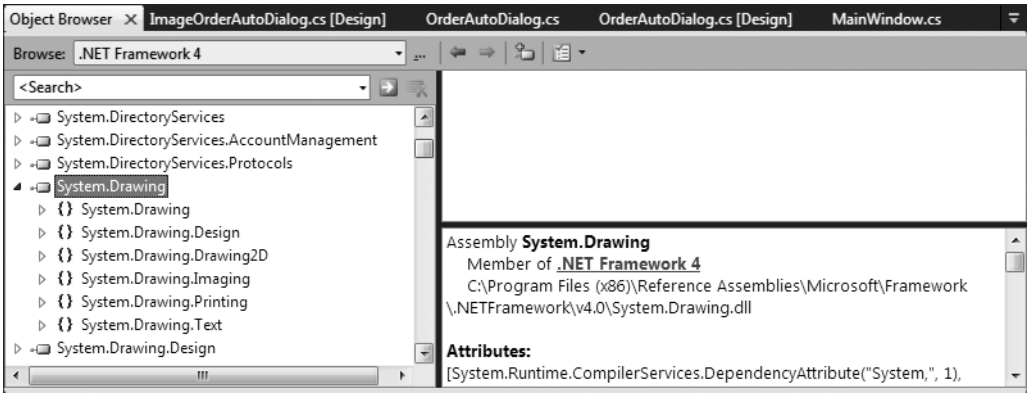

Рис. А.20. Пространства имен сборки `System.Drawing.dll`

На заметку! Запомните, что в WPF имеется собственная подсистема графической визуализации и API-интерфейс; GDI+ необходим только в приложениях Windows Forms.

В табл. А.10 приведены высокоуровневые пояснения ролей основных пространств имен GDI+.

Таблица А.10. Основные пространства имен GDI+

Пространство имен	Описание
`System.Drawing`	Основное пространство имен GDI+, которое определяет многочисленные типы для обычного отображения (шрифты, перья и простые кисти), а также тип `Graphics`
`System.Drawing.Drawing2D`	Содержит типы для более сложных двумерных и векторных графических операций (градиентные кисти, кончики пера и геометрические преобразования)
`System.Drawing.Imaging`	Определяет типы для работы с графическими изображениями (изменение палитры, извлечение метаданных изображения, манипулирование метафайлами и т.д.)
`System.Drawing.Printing`	Определяет типы для визуализации изображений на печатных страницах, взаимодействия с принтерами и форматирования общего вида для задания печати
`System.Drawing.Text`	Позволяет работать с коллекциями шрифтов

Пространство имен System.Drawing

Большинство типов, необходимых при программировании приложений GDI+, находятся в пространстве имен System.Drawing. В нем можно найти классы, которые представляют изображения, кисти, перья и шрифты. Кроме того, System.Drawing определяет несколько вспомогательных типов, таких как Color, Point и Rectangle. В табл. А.11 приведены некоторые (но не все) основные типы.

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

Тип	Описание
Bitmap	Инкапсулирует данные изображения (*.bmp или другие)
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 нужна для снижения или устранения мерцания вследствие перерисовки поверхности элемента
Color SystemColors	Определяют ряд статических свойств только для чтения, которые позволяют получить конкретные цвета для создания различных перьев и/или кистей
Font FontFamily	Тип Font инкапсулирует характеристики данного шрифта (имя, плотность, наклон и размер в пунктах). FontFamily предоставляет абстракцию для группы шрифтов схожего вида, но с некоторыми расхождениями в стиле
Graphics	Основной класс, представляющий поверхность для рисования, а также несколько методов для визуализации текста, изображений и геометрических фигур
Icon SystemIcons	Представляют специальные значки и набор стандартных значков, поддерживаемых системой
Image ImageAnimator	Image — абстрактный базовый класс, содержащий функциональность для типов Bitmap, Icon и Cursor. Класс ImageAnimator предоставляет возможность прохода в цикле по нескольким производным от Image типам через заданный промежуток времени
Pen Pens SystemPens	Объекты, используемые для рисования прямых и кривых линий. Тип Pens определяет несколько статических свойств, которые возвращают новый объект Pen заданного цвета
Point PointF	Структуры, представляющие отображение координат (*x*, *y*) на соответствующие целые и дробные значения
Rectangle RectangleF	Структуры, представляющие размеры прямоугольника (также отображаются на соответствующие целые и дробные значения)
Size SizeF	Структуры, представляющие высоту и ширину (также отображаются на соответствующие целые и дробные значения)
StringFormat	Инкапсулирует различные характеристики текстовой компоновки (например, выравнивание и промежутки между строками)
Region	Описывает внутреннюю область геометрической фигуры, составленной из прямоугольников и ломаных линий

Роль типа Graphics

Класс System.Drawing.Graphics служит шлюзом для функциональности визуализации GDI+. Он не только представляет поверхность для рисования (поверхность формы, элемента управления или области памяти), но и определяет десятки членов, позволяющих выводить текст, изображения (например, значки и битовые изображения), а также различные геометрические фигуры. Частичный список членов класса приведен в табл. А.12.

Таблица А.12. Избранные члены класса Graphics

Метод	Описание
FromHdc() FromHwnd() FromImage()	Статические методы, определяющие способ получения объекта Graphics из данного изображения (к примеру, значка или битового изображения) или графического элемента
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() используют объекты Pen из GDI+
FillEllipse() FillPie() FillPolygon() FillRectangle() FillPath()	Методы для заполнения внутренней области заданной геометрической фигуры. Все методы FillXXX() используют объекты Brush из GDI+

Экземпляры класса Graphics невозможно создать напрямую с помощью ключевого слова new, т.к. у этого класса нет открытых конструкторов. А как же получить объект Graphics? Ищите ответ ниже.

Получение объекта Graphics с помощью события Paint

Чаще всего объект Graphics получают с помощью окна Properties в Visual Studio: в нем можно обработать событие Paint для окна, на котором нужно выполнить визуализацию. Это событие определено через делегат PaintEventHandler, который может указывать на любой метод, принимающий в качестве первого параметра System.Object, а в качестве второго — PaintEventArgs.

Параметр PaintEventArgs содержит объект Graphics, необходимый для визуализации на поверхности формы. Для примера создайте новый проект Windows Forms Application по имени PaintEventApp. Затем с помощью Solution Explorer измените имя первоначального файла Form.cs на MainWindow.cs и в окне Properties создайте обработчик события Paint. При этом будет сгенерирована следующая заглушка:

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

Теперь, когда имеется обработчик события `Paint`, может возникнуть вопрос: когда инициируется это событие? Событие `Paint` возникает, когда окно становится *запорченным* — т.е. когда изменяется его размер, когда его перестает загораживать (частично или полностью) другое окно или когда оно было свернуто, а затем развернуто. Во всех этих случаях платформа .NET автоматически вызывает обработчик события `Paint`. Рассмотрим следующую 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 другой метод с именем, начинающимся на `Fill`) требует в качестве первого параметра тип, производный от `Brush`. В принципе, можно самостоятельно создать любое количество интересных объектов кистей из пространства имен `System.Drawing.Drawing2D` (например, `HatchBrush` и `LinearGradientBrush`), но вдобавок имеется вспомогательный класс `Brushes`, который предлагает удобный доступ к множеству типов кистей со сплошным цветом.

Затем вызывается метод `DrawString()`, которому в первом параметре передается визуализируемая строка. Для этого в GDI+ имеется тип `Font`, который представляет не только имя шрифта для отображения текстовых данных, но и характеристики этого шрифта, например, размер в пунктах (в данном случае 30). Методу `DrawString()` также нужен тип `Brush` — ведь с точки зрения GDI+ строка “Hello GDI+” представляет собой просто набор геометрических фигур, которые нужно вывести на экране. В конце вызывается метод `DrawLine()`, который выводит прямую линию с помощью специального типа `Pen` шириной 10 пикселей. Результат работы этого кода показан на рис. А.21.

На заметку! В приведенном выше коде объект `Pen` освобождается явным образом. Как правило, при непосредственном создании объекта типа GDI+, который реализует интерфейс `IDisposable`, сразу после окончания работы с данным объектом необходимо вызвать метод `Dispose()`. Это позволяет освободить занятые ресурсы как можно раньше. Если не сделать этого, то ресурсы, в конце концов, будут освобождены с помощью сборщика мусора, но в недетерминированной манере.

Рис. А.21. Простые операции визуализации GDI+

Объявление недействительной клиентской области формы

Во время работы приложения Windows Forms может понадобиться явно инициировать в коде событие `Paint`, а не ждать, пока окно станет *естественно запорченным* в результате действий пользователя. К примеру, 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. Понятно, что если ожидать естественного загрязнения формы, то пользователь не увидит никаких изменений, пока не изменит размер окна или не прикроет его другим окном. Чтобы программным образом выполнить перерисовку окна, необходимо вызвать унаследованный метод `Invalidate()`:

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

Метод `Invalidate()` имеет несколько перегруженных версий. Это позволяет указать конкретную прямоугольную часть формы, которую нужно перерисовать, чтобы не выполнять перерисовку всей клиентской области (что происходит по умолчанию). Если требуется обновить только небольшой прямоугольник в верхнем левом углу клиентской области, то можно написать примерно 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 этого вводного обзора API-интерфейсов Windows Forms и GDI+ мы построим полное приложение с графическим пользовательским интерфейсом, в котором будут продемонстрированы многие описанные ранее приемы. Это будет примитивная 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

Вначале создайте новое приложение Windows Forms 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. A.22.

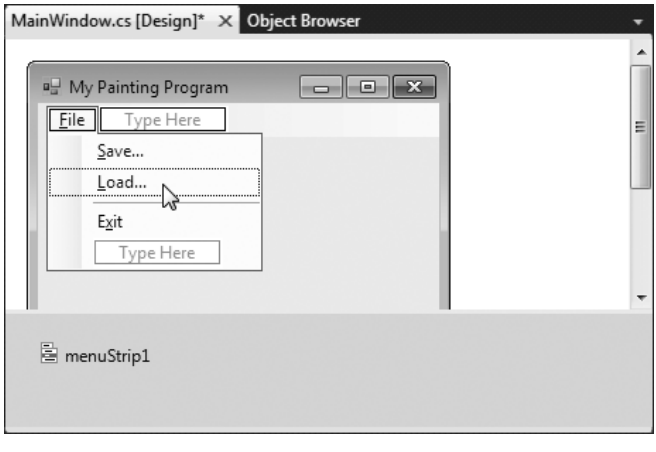

Рис. A.22. Система меню File

На заметку! Указание в качестве пункта меню одного дефиса (-) позволяет вставлять разделители в систему меню.

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

Наконец, обработайте событие `Click` для каждого из построенных подпунктов меню. Все эти обработчики будут создаваться по мере выполнения примера, а пока можно заполнить обработчик для пункта File Exit с помощью вызова метода `Close()`:

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

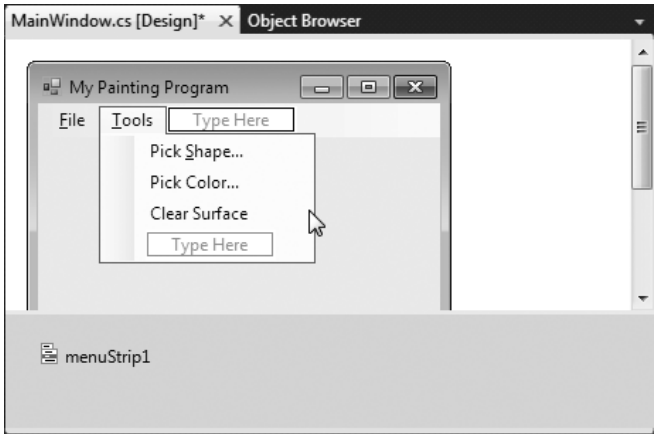

Рис. А.23. Система меню Tools

Определение типа ShapeData

Как упоминалось выше, это приложение должно позволять конечным пользователям выбирать одну из двух предопределенных фигур и ее цвет. 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 мы сделаем это с помощью автоматических свойств C#. Вначале добавьте в проект ShapeData.cs новый класс и реализуйте необходимый тип следующим образом:

```
[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; }
}
```

Здесь класс ShapeData использует автоматические свойства, инкапсулирующие различные типы данных, два из которых (Point и Color) определены в пространстве имен System.Drawing, поэтому не забудьте импортировать это пространство имен в файл кода. Обратите внимание, что у данного типа имеется атрибут [Serializable]. Ниже тип MainWindow будет сконфигурирован для поддержки списка типов ShapeData, который будет храниться с помощью служб сериализации объектов.

Определение типа ShapePickerDialog

Чтобы пользователь мог выбрать круг или прямоугольник, можно создать простое диалоговое окно с именем ShapePickerDialog (вставьте новый тип Form). Кроме традиционных кнопок OK и Cancel (с присвоенными соответствующими значениями DialogResult), в диалоговом окне будет находиться один элемент GroupBox, содержащий два объекта RadioButton: radioButtonCircle и radioButtonRect. На рис. А.24 приведен один из вариантов построения.

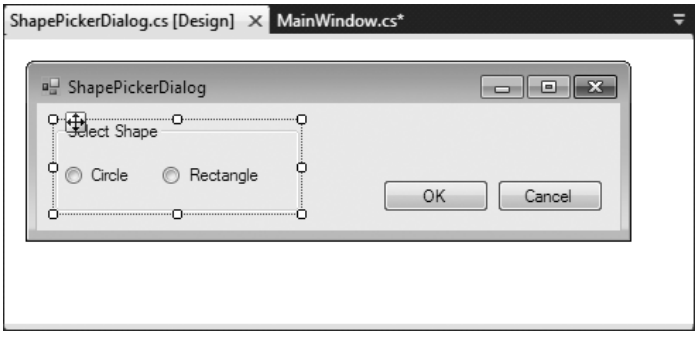

Рис. А.24. Тип `ShapePickerDialog`

Теперь откройте окно кода для данного диалогового окна. Для этого щелкните правой кнопкой мыши на поверхности визуального конструктора форм и выберите в контекстном меню пункт View Code (Просмотреть код). В пространстве имен `MyPaintProgram` объявите перечисление по имени `SelectedShape`, определяющее имена возможных фигур:

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

Теперь модифицируйте текущий класс `ShapePickerDialog`, как описано ниже.

- Добавьте автоматическое свойство типа `SelectedShape`. Вызывающий метод может использовать это свойство для определения, какую фигуру следует визуализировать.
- Создайте обработчик события `Click` для кнопки OK с помощью окна Properties.
- Реализуйте этот обработчик события, чтобы он определял, выбран ли переключателя `radioButtonCircle` (с помощью свойства `Checked`). Если это так, свойство `SelectedShape` должно быть установлено 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. Осталось реализовать обработчики событий `Click` для остальных пунктов меню главного окна.

Добавление инфраструктуры в тип MainWindow

Вернемся к построению главного окна и добавим в нашу форму три новых переменных-члена. Эти переменные позволят отслеживать выбранную фигуру (с помощью перечисления SelectedShape), выбранный цвет (представленный переменной-членом System.Drawing.Color) и все визуализированные изображения, которые хранятся 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 хранятся все ShapeData.
  private List<ShapeData> shapes = new List<ShapeData>();
  ...
}
```

Далее создадим обработчики событий MouseDown и Paint для данного типа, производного от Form, с помощью окна Properties. Мы реализуем их на следующем шаге, а пока IDE-среда просто сгенерирует следующие заглушки:

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

Реализация функциональности меню Tools

Чтобы пользователь мог 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 основании выбора пользователя устанавливать значение соответствующей переменной-члена:

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

Чтобы позволить пользователю устанавливать значение переменной-члена currentColor, необходимо реализовать обработчик события Click для пункта меню Tools Pick Color... (Сервис⇨Выбрать цвет...), в котором применяется стандартный 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;
  }
}
```

Если теперь запустить полученную программу и выбрать пункт меню Tools Pick Color..., то откроется диалоговое окно, показанное на рис. А.25.

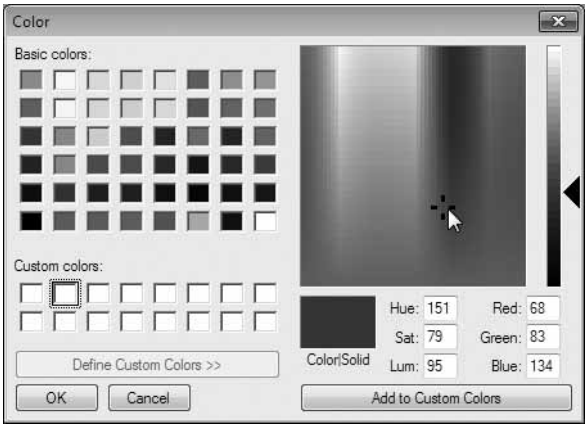

Рис. А.25. Стандартный тип `ColorDialog`

И в завершение необходимо реализовать обработчик пункта меню Tools Clear Surface, который очищает содержимое переменной-члена `List<T>` и программно генерирует событие `Paint` с помощью вызова метода `Invalidate()`:

```
private void clearSurfaceToolStripMenuItem_Click(object sender, EventArgs e)
{
  shapes.Clear();

  // Инициировать событие Paint.
  Invalidate();
}
```

Захват и визуализация графического вывода

Поскольку вызов `Invalidate()` инициирует событие `Paint`, понадобится добавить код в обработчик события `Paint`. В нем нужно реализовать цикл по всем элементам (пока пустого) списка `List<T>` и визуализировать в текущем положении курсора мыши окружность или прямоугольник. Для этого сначала нужно реализовать обработчик события `MouseDown` и вставить новый объект `ShapeData` в обобщенный список `List<T>`, учитывая выбранные пользователем цвет, вид и текущее положение 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();
}
```

В этот момент можно реализовать 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, должна появиться возможность визуализации любого количества фигур произвольных цветов (рис. А.26).

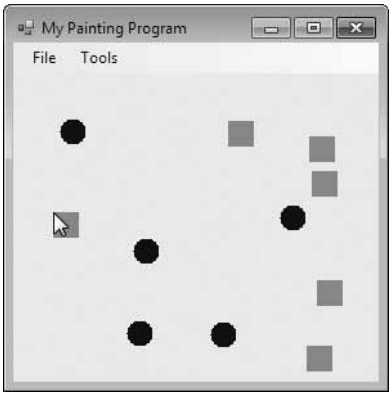

Рис. А.26. Приложение `MyPaintProgram` в действии

Реализация логики сериализации

И последнее, что потребуется сделать в проекте — реализовать обработчики событий для пунктов меню File Save... и File Load.... Поскольку класс `ShapeData` снабжен атрибутом `[Serializable]` (а класс `List<T>` является сериализируемым сам по себе), текущие графические данные можно сохранить с помощью имеющегося в Windows Forms типа `SaveFileDialog`. Но перед этим нужно указать в директивах `using`, что будут использоваться пространства имен `System.Runtime.Serialization.Formatters.Binary` и `System.IO`:

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

Теперь измените обработчик пункта меню File Save... следующим образом:

```
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,
    // 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 File Load... открывает выбранный файл и десериализирует данные в переменную-член `List<T>`. Для этого применяется имеющийся в Windows Forms тип `OpenFileDialog`:

```
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. Важно отметить, что диалоговые окна `SaveFileDialog` и `OpenFileDialog` имеют свойство `Filter`, которому можно присвоить не очень понятное строковое значение, представляющее собой фильтр. Этот фильтр управляет рядом параметров для диалоговых окон сохранения и открытия файлов — в частности, файловым расширением (`*.shapes`). Свойство `FileName` позволяет указать стандартное имя сохраняемого файла — `MyShapes` в этом примере.

На этом пример приложения для рисования фигур завершен. Теперь должна быть возможность сохранения графических данных в любом количестве файлов `*.shapes` и загрузки их снова. При желании данную программу можно усовершенствовать: добавить дополнительные фигуры или позволить пользователю управлять размером и формой фигур, а также выбирать формат сохраняемых данных (например, двоичный, XML или SOAP).

Резюме

В этом приложении был рассмотрен процесс построения традиционных пользовательских приложений с помощью API-интерфейсов Windows Forms и GDI+, которые входят в состав .NET Framework, начиная с версии 1.0. Минимальное приложение Windows Forms состоит из типа, расширяющего `Form`, и метода `Main()`, который взаимодействует с типом `Application`.

Если требуется заполнить формы элементами пользовательского интерфейса (например, системами меню и элементами управления вводом данных), то это можно сделать, добавляя новые объекты в унаследованную коллекцию `Controls`. В этом приложении также было показано, как реагировать на события мыши, клавиатуры и визуализации. Попутно вы ознакомились с типом `Graphics` и множеством способов генерации графических данных во время выполнения.

Как уже было сказано, API-интерфейс Windows Forms (в какой-то мере) вытеснен API-интерфейсом WPF, который появился в версии .NET 3.0. Конечно, инфраструктура WPF удобна для создания насыщенных пользовательских интерфейсов, однако API-интерфейс Windows Forms остается простейшим (и зачастую наиболее прямым) способом создания стандартных бизнес-приложений, приложений для внутреннего использования и простых утилит конфигурирования. По этим причинам в ближайшие годы Windows Forms будет входить в состав библиотек базовых классов .NET.