ДОПОЛНИТЕЛЬНАЯ ГЛАВА 2

# Построение улучшенных окон с помощью System.Windows.Forms

Вы уже должны обладать солидными знаниями языка программирования C#, а также основ архитектуры .NET. Хотя можно воспользоваться приобретенными знаниями и приступить к построению следующего поколения консольных приложений, скорее всего, интереснее будет спроектировать привлекательный графический пользовательский интерфейс (GUI), чтобы позволить пользователям взаимодействовать с вашей системой.

Эта глава нацелена на ознакомление с процессом построения традиционного настольного приложения на базе экранных форм. Здесь вы научитесь строить высокостилизованное главное окно, используя классы `Form` и `Application`. В главе также иллюстрируется получение пользовательского ввода и реакцию на него (т.е. обработка событий мыши и клавиатуры) в контексте настольной среды GUI. И, наконец, вы научитесь конструировать системы меню, панели инструментов, линейки состояния и приложения с многодокументным интерфейсом (MDI), как вручную, так с помощью средств дизайна, встроенных в Visual Studio.

## Обзор пространства имен System.Windows.Forms

Подобно любому пространству имен, `System.Windows.Forms` состоит из различных классов, структур, делегатов, интерфейсов и перечислений. Хотя разница внешнего вида между консольным пользовательским интерфейсом (CUI) и графическим пользовательским интерфейсом (GUI) кажется на первый взгляд разительной, на самом деле процесс построения приложения Windows Forms не представляет собой ничего свыше манипуляций с новым набором типов на основе уже известного синтаксиса C#. На самом высоком уровне сотни типов из пространства имен `System.Windows.Forms` можно разделить на следующие обширные категории.

- *Инфраструктура ядра.* Это типы, представляющие основные операции программы .NET Forms (`Form`, `Application` и т.п.) и различные типы для облегчения взаимодействия с унаследованными элементами управления ActiveX.

- *Элементы управления.* Это типы, используемые для создания развитых пользовательских интерфейсов (Button, MenuStrip, ProgressBar, DataGridView и т.д.), которые унаследованы от базового класса Control. Элементы управления конфигурируемы во время проектирования и видимы (по умолчанию) во время выполнения.
- *Компоненты.* Это типы, не унаследованные от базового класса Control, но предоставляющие визуальные средства программе .NET Forms (ToolTip, ErrorProvider и т.д.). Многие компоненты (такие как Timer) не видимы во время выполнения, но могут быть конфигурированы во время проектирования.
- *Общие диалоговые окна.* Windows Forms предлагает множество готовых диалоговых окон для часто выполняемых операций (OpenFileDialog, PrintDialog и т.п.). Как можно было надеяться, всегда можно построить свое собственное диалоговое окно, если стандартное не удовлетворяет существующим потребностям.

Учитывая, что общее количество типов в System.Windows.Forms превышает сотню, было бы излишним перечислять каждый член семейства Windows Forms. Однако в качестве фундамента в табл. 2.1 перечислены некоторые из основных типов .NET 2.0 из пространства имен System.Windows.Forms (все подробности ищите в документации по .NET Framework 2.0 SDK).

**Таблица 2.1. Основные типы пространства имен System.Windows.Forms**

| Классы | Назначение |
|---|---|
| Application | Этот класс инкапсулирует операции времени выполнения приложения Windows Forms. |
| Button, CheckBox, ComboBox, DateTimePicker, ListBox, LinkLabel, MaskedTextBox, MonthCalendar, PictureBox, TreeView | Эти классы (в дополнение ко многим другим) соответствуют различным виджетам GUI. |
| FlowLayoutPanel, TableLayoutPanel | .NET 2.0 теперь предлагает различные “диспетчеры компоновки”, которые автоматически размещают элементы управления форм при изменении размера последних. |
| Form | Этот тип представляет главное окно, диалоговое окно или дочернее окно MDI приложения Windows Forms. |
| ColorDialog, OpenFileDialog, SaveFileDialog, FontDialog, PrintPreviewDialog, FolderBrowserDialog | Это различные стандартные диалоговые окна для часто применяемых операций GUI. |
| Menu, MainMenu, MenuItem, ContextMenu, MenuStrip, ContextMenuStrip | Эти типы используются для построения систем меню верхнего уровня и контекстных меню. Появившиеся в .NET 2.0 элементы управления MenuStrip и ContextMenuStrip позволяют строить меню, которые могут содержать традиционные раскрывающиеся пункты меню, а также другие элементы управления (текстовые поля, раскрывающиеся списки и т.д.). |
| StatusBar, Splitter, ToolBar, ScrollBar, StatusStrip, ToolStrip | Эти типы используются для оснащения форм распространенными дочерними элементами управления. |

**На заметку!** В дополнение к System.Windows.Forms, сборка System.Windows.Forms.dll определяет пространства имен, касающиеся GUI. По большей части эти дополнительные типы используются внутренне механизмом Windows Forms и/или инструментами дизайна Visual Studio. Учитывая этот факт, мы здесь сосредоточимся на пространстве имен System.Windows.Forms.

# Работа с типами Windows Forms

При построении приложения Windows Forms можно написать весь необходимый код вручную (в простом текстовом редакторе вроде Notepad) и передать полученные в результате файлы *.cs компилятору C# вместе с флагом /target:winexe. Построение некоторых приложений Windows Forms вручную не только даст неоценимый опыт, но также поможет лучше понять код, генерируемый различными дизайнерами, которые имеются в составе интегрированных сред разработки для .NET.

Чтобы предоставить реальное представление о базовом процессе построения приложения Windows Forms, в начальных примерах этой главы графические дизайнеры применяться не будут. Почувствовав себя уверенно при построении приложений Windows Forms вручную, вы сможете перейти к использованию различных инструментов дизайна, предлагаемых Visual Studio.

## Построение главного окна вручную

Чтобы приступить к изучению программирования Windows Forms, построим минимальную форму окна с самого начала. Создайте новую папку на жестком диске (например, C:\MyFirstWindow) и внутри нее новый файл MainWindow.cs с помощью какого-нибудь текстового редактора.

В мире Windows Forms класс Form используется для представления любого окна приложения. Сюда входит и главное окно верхнего уровня в приложении с однодокументным интерфейсом (SDI), немодальные и модальные диалоговые окна, а также родительские и дочерние окна приложений многодокументного интерфейса (MDI). Когда вы заинтересованы в создании и отображении главного окна вашей программы, нужно выполнить два обязательных шага.

1. Унаследовать новый класс от System.Windows.Forms.Form.
2. Сконфигурировать метод Main() приложения для вызова Application.Run() с передачей ему экземпляра вашего типа-наследника Form в качестве аргумента.

Учитывая сказанное, модифицируйте файл MainWindows.cs следующим образом:

```
using System;
using System.Windows.Forms;

namespace MyWindowsApp
{
  public class MainWindow : Form
  {
    // Запустить приложение, указав главное окно.
    static void Main(string[] args)
    {
      Application.Run(new MainWindow());
    }
  }
}
```

В дополнение к всегда присутствующей сборке mscorlib.dll, приложение Windows Forms нуждается в ссылке на сборки System.dll и System.Windows.Forms.dll. Вспомните, что ответный файл C# по умолчанию (csc.rsp) инструктирует csc.exe автоматически включать эти сборки в процесс компиляции, так что ничего больше делать не потребуется. Также вспомните, что опция /target:winexe csc.exe инструктирует компилятор генерировать исполняемую программу Windows.

---

**На заметку!** С технической точки зрения приложение Windows можно построить в командной строке, используя опцию /target:exe; однако при этом вы обнаружите, что командное окно окажется на заднем плане (и останется там до тех пор, пока вы не закроете главное окно). Когда указывается /target:winexe, исполняемая программа запускается как родное приложение Windows Forms (без подвешивания командного окна).

---

Для компиляции файла кода C# откройте командное окно Visual Studio и введите следующую команду:

```
csc /target:winexe *.cs
```

На рис. 2.1 показан результат тестового запуска.

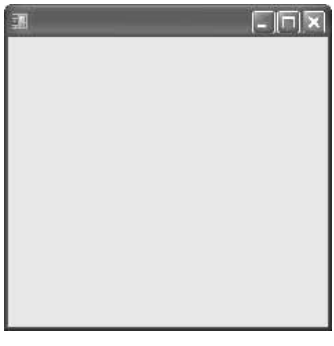

**Рис. 2.1.** Простое главное окно в стиле Windows Forms

Понятно, что пока полученная форма не представляет никакого интереса. Но простым наследованием от Form вы получаете главное окно с возможностью разворачивания, сворачивания, изменения размера и закрытия (с системной пиктограммой по умолчанию). В отличие от других программных интерфейсов GUI, которые, возможно, приходилось использовать в прошлом (в частности, Microsoft Foundation Classes), здесь нет необходимости писать сотни строк инфраструктуры кодирования (фреймов, документов, представлений, приложений иди карт сообщений). В отличие от приложения на C, основанного на Win32 API, не понадобится вручную реализовывать процедуры WinProc() или WinMain(). На платформе .NET все эти детали инкапсулированы внутри типов Form и Application.

## Обеспечение разделения труда

В настоящее время класс MainWindow определяет метод Main() непосредственно внутри своего определения. Если хотите, можете создать второй статический класс (Program), который будет отвечать за запуск главного окна, оставляя производному от Form классу отображение самого окна:

```
namespace MyWindowsApp
{
  // Главное окно.
  public class MainWindow : Form { }

  // Объект приложения.
  public static class Program
  {
    static void Main(string[] args)
    {
      // Не забыть использовать System.Windows.Forms!
      Application.Run(new MainWindow());
    }
  }
}
```

Сделав так, вы последуете одному из принципов ООП, именуемому *разделение труда* (separation of concerns). Просто говоря, это правило дизайна ООП устанавливает, что класс должен отвечать за минимально возможный объем работ. Учитывая, что начальный класс разделен на два уникальных класса, мы отделяем форму от класса, создающего ее. В результате мы получаем более переносимое окно, поскольку оно может быть легко переброшено в другой проект, и при этом не придется беспокоиться о дополнительном багаже специфичного для проекта метода Main().

**Исходный код.** Проект MyFirstWindow находится в подкаталоге Bonus Chapter 2.

# Роль класса Application

Класс Application определяет многочисленные статические методы, которые позволяют управлять различными низкоуровневыми нюансами поведения приложения Windows Forms. Например, класс Application определяет набор событий, которые позволяют реагировать на такие ситуации, как останов приложения и обработка времени ожидания (простоя). Вдобавок к Run() есть еще несколько методов, о которых следует знать.

- DoEvents(). Обеспечивает приложению возможность обрабатывать сообщения, находящиеся в очереди сообщений во время длительной операции.
- Exit(). Завершает приложение Windows и выгружает принимающий его домен приложений (AppDomain).
- EnableVisualStyles(). Конфигурирует приложение для поддержки визуальных стилей Windows XP. Обратите внимание, что для разрешения стилей XP этот метод должен вызываться перед загрузкой главного окна посредством Application.Run().

Класс Application также определяет ряд свойств, многие из которых по природе доступны только для чтения. Как видно в табл. 2.2, большинство из этих свойств представляют особенности "уровня приложения", такие как имя компании, номер версии и т.п. Фактически, учитывая, что вы уже знаете об атрибутах уровня сборки, многие из этих свойств покажутся знакомыми.

**Таблица 2.2. Основные свойства типа Application**

| Свойство | Назначение |
|---|---|
| CompanyName | Извлекает значение атрибута [AssemblyCompany] уровня сборки. |
| ExecutablePath | Получает путь к исполняемому файлу. |
| ProductName | Извлекает значение атрибута [AssemblyProduct] уровня сборки. |
| ProductVersion | Извлекает значение атрибута [AssemblyVersion] уровня сборки. |
| StartupPath | Извлекает путь к исполняемому файлу, запустившему приложение. |

И, наконец, класс Application определяет различные статические события, некоторые из них перечислены ниже.

- ApplicationExit. Происходит перед завершением приложения.
- Idle. Происходит, когда цикл сообщений приложения завершил обработку текущего набора сообщений и готов войти в состояние ожидания (поскольку в очереди в данный момент нет новых сообщений для обработки).
- ThreadExit. Случается перед остановом потока приложения.

# Упражнения с классом Application

Чтобы проиллюстрировать некоторую функциональность класса Application, давайте расширим текущий класс MainWindows следующими возможностями:

- отображение избранных атрибутов уровня сборки;
- обработка статического события ApplicationExit.

Первая задача — использование избранных свойств класса Application для отображения некоторых атрибутов уровня сборки. Для начала добавьте следующие атрибуты в файл MainWindow.cs (обратите внимание на добавление ссылки using на пространство имен System.Reflection):

```
using System;
using System.Windows.Forms;
using System.Reflection;
// Некоторые атрибуты этой сборки.
[assembly:AssemblyCompany("Intertech Training")]
[assembly:AssemblyProduct("A Better Window")]
[assembly:AssemblyVersion("1.1.0.0")]
namespace MyWindowsApp
{
  ...
}
```

Вместо ручного отображения атрибутов [AssemblyCompany] и [AssemblyProduct] класс Application делает это автоматически посредством различных статических свойств. Для иллюстрации реализуйте конструктор по умолчанию MainForm:

```
public class MainWindow : Form
{
  public MainWindow()
  {
    MessageBox.Show(Application.ProductName,
      string.Format("This app brought to you by {0}", Application.CompanyName));
  }
}
```

Запустив это приложение, вы увидите окно сообщения, отображающее различные части информации (рис. 2.2).


**Рис. 2.2.** Чтение атрибутов через тип Application

Теперь давайте заставим форму реагировать на событие ApplicationExit. Когда вы захотите отреагировать на событие внутри приложения Windows Forms, то будете приятно удивлены, обнаружив, что описанный ранее синтаксис событий используется для обработки событий GUI. Таким образом, если необходимо перехватить статическое событие ApplicationExit, просто зарегистрируйте обработчик события, используя операцию +=:

```
public class MainForm : Form
{
  public MainForm()
  {
    ...
    // Перехватить событие ApplicationExit.
    Application.ApplicationExit += new EventHandler(MainWindow_OnExit);
  }
  private void MainWindow_OnExit(object sender, EventArgs evArgs)
  {
    MessageBox.Show(string.Format("Form version {0} has terminated.",
      Application.ProductVersion));
  }
}
```

## Делегат System.EventHandler

Обратите внимание, что событие ApplicationExit работает в сочетании с делегатом System.EventHandler. Этот делегат должен указывать на методы со следующей сигнатурой:

```
delegate void EventHandler(object sender, EventArgs e);
```

System.EventHandler — наиболее примитивный делегат, используемый для обработки событий внутри Windows Forms, но для других событий существует много его вариаций. Что касается EventHandler, то первый параметр присваиваемого ему метода имеет тип System.Object и представляет объект, отправивший событие. Второй параметр типа EventArgs (или его наследника) содержит всю важную информацию, касающуюся текущего события.

---

**На заметку!** EventArgs — базовый класс для многочисленных производных типов, содержащих информацию о семействе взаимосвязанных событий. Например, события мыши работают с параметром типа MouseEventArgs, содержащим такие детали, как координаты курсора мыши (*x*, *y*). Многие события клавиатуры работают с типом KeyEventArgs, содержащим подробности относительно текущих нажатий клавиш, и т.д.

---

В любом случае, если вы перекомпилируете и запустите приложение, то по завершении работы приложения увидите окно сообщения.

**Исходный код.** Проект AppClassExample находится в подкаталоге Bonus Chapter 2.

# Внутренняя структура Form

Теперь, когда вы понимаете роль типа Application, следующей задачей будет изучение функциональности самого класса Form. Не удивительно, что класс Form наследует значительный объем функциональности от своих родительских классов. На рис. 2.3 показана цепочка наследования (включая набор реализованных интерфейсов) производного от Form типа в окне Object Browser среды Visual Studio.

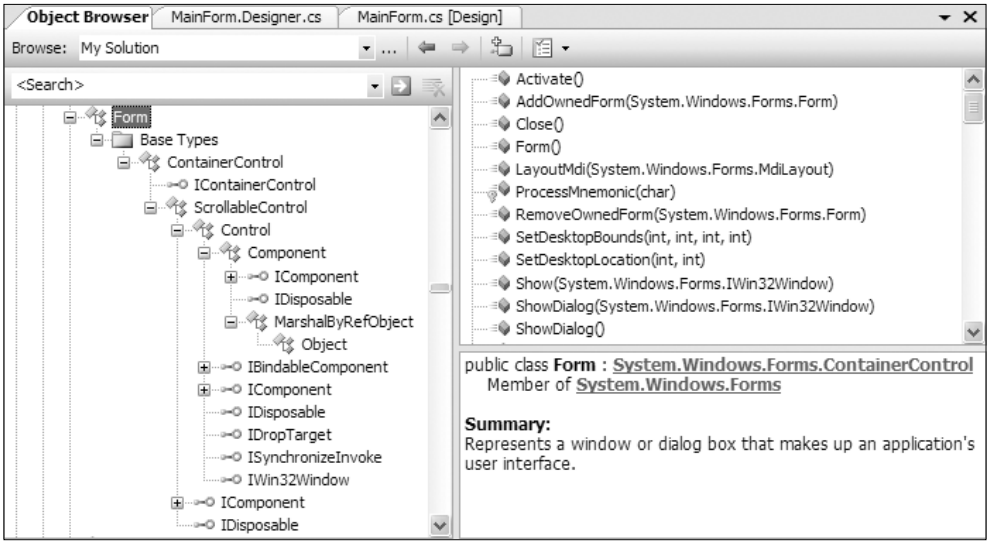


**Рис. 2.3.** Цепочка наследования типа Form

Хотя полная "родословная" типа Form включает многочисленные базовые классы и интерфейсы, знайте, что вы *не* обязаны знать назначение всех и каждого члена из каждого родительского класса и реализованного интерфейса, чтобы стать профессиональным разработчиком Windows Forms. Фактически, большинство членов (в частности, свойств и событий), которые вы будете применять повседневно, легко устанавливаются с помощью окна Properties (Свойства) интегрированной среды разработки Visual Studio. Прежде чем переходить к рассмотрению некоторых специфических членов, унаследованных от родительских классов, ознакомьтесь с табл. 2.3, в которой описаны роли базовых классов Form.

Как вы можете догадаться, детальное описание каждого члена каждого класса в цепочке наследования Form само по себе потребовало бы отдельной книги. Однако важно понимать поведение, поддерживаемое типами Control и Form. Уделите некоторое время на изучение подробностей устройства каждого класса в документации по .NET Framework 2.0 SDK.

**Таблица 2.3. Базовые классы в цепочке наследования** `Form`

| Родительский класс | Назначение |
|---|---|
| `System.Object` | Подобно любому классу в .NET, класс `Form` "является" `Object`. |
| `System.MarshalByRefObject` | Вспомните из описания .NET Remoting, что типы, унаследованные от этого класса, доступны удаленно через *ссылку на* удаленный тип (а не его локальную копию). |
| `System.ComponentModel.Component` | Этот класс представляет реализацию по умолчанию интерфейса `IComponent`. В мире .NET компонент — это тип, поддерживающий редактирование во время дизайна, но не обязательно видимый во время выполнения. |
| `System.Windows.Forms.Control` | Этот класс определяет общие члены пользовательского интерфейса для всех элементов управления Windows Forms, включая сам тип `Form`. |
| `System.Windows.Forms.ScrollableControl` | Этот класс определяет поддержку поведения автоматической прокрутки. |
| `System.Windows.Forms.ContainerControl` | Этот класс предоставляет функциональность управления фокусом для элементов управления, которые могут служить контейнерами для других элементов управления. |
| `System.Windows.Forms.Form` | Этот класс представляет любую форму, дочернее окно MDI или диалоговое окно. |

## Функциональность класса `Control`

Класс `System.Windows.Forms.Control` устанавливает общее поведение, необходимое любому типу GUI. Основные члены `Control` позволяют конфигурировать размер и положение элемента управления, получать ввод с клавиатуры и мыши, получать и устанавливать фокус и видимость члена, и т.д. В табл. 2.4 перечислены некоторые интересные свойства, сгруппированные по функциональности.

**Таблица 2.4. Основные свойства типа `Control`**

| Свойства | Назначение |
|---|---|
| `BackColor`, `ForeColor`, `BackgroundImage`, `Font`, `Cursor` | Эти свойства определяют основной пользовательский интерфейс элемента управления (цвета, шрифт текста, курсор мыши, отображаемый, когда он находится над элементом, и т.д.). |
| `Anchor`, `Dock`, `AutoSize` | Эти свойства управляют позиционированием элемента управления в контейнере. |
| `Top`, `Left`, `Bottom`, `Right`, `Bounds`, `ClientRectangle`, `Height`, `Width` | Эти свойства специфицируют текущие размеры элемента управления. |
| `Enabled`, `Focused`, `Visible` | Эти свойства возвращают булевские значения, показывающие состояние текущего элемента управления. |

*Окончание табл. 2.4*

| Свойства | Назначение |
|---|---|
| ModifierKeys | Это статическое свойство проверяет текущее состояние модифицирующих клавиш (<Shift>, <Ctrl> и <Alt>) и возвращают его в типе Keys. |
| MouseButtons | Это статическое свойство проверяет текущее состояние кнопок мыши (левой, правой и средней) и возвращает это состояние с использованием типа MouseButtons. |
| TabIndex, TabStop | Эти свойства используются для конфигурирования порядка обхода элементов управления по клавише <Tab>. |
| Opacity | Это свойство определяет прозрачность элемента управления (0.0 — полностью прозрачный, 1.0 — полностью непрозрачный) |
| Text | Это свойство указывает строковые данные, ассоциированные с элементом управления. |
| Controls | Это свойство позволяет обратиться к строго типизованной коллекции (ControlsCollection), содержащей любые дочерние элементы внутри текущего элемента управления. |

Несложно догадаться, что класс Control также определяет ряд событий, которые можно перехватывать, чтобы отследить действия мыши, клавиатуры, рисования и перетаскивания (помимо прочих). В табл. 2.5 перечислены некоторые интересные события, сгруппированные по функциональности.

**Таблица 2.5. События типа Control**

| События | Назначение |
|---|---|
| Click, DoubleClick, MouseEnter, MouseLeave, MouseDown, MouseUp, MouseMove, MouseHover, MouseWheel | Различные события, позволяющие взаимодействовать с мышью. |
| KeyPress, KeyUp, KeyDown | Различные события, позволяющие взаимодействовать с клавиатурой. |
| DragDrop, DragEnter, DragLeave, DragOver | Различные события, используемые для отслеживания действия перетаскивания. |
| Paint | Событие, позволяющее взаимодействовать со службами визуализации графики GDI+. |

И, наконец, базовый класс Control также определяет методы, позволяющие взаимодействовать с любым типом-наследником Control. По мере знакомства с методами типа Control вы заметите, что значительное их число имеет префикс On, за которым следует имя определенного события (OnMouseMove, OnKeyUp, OnPaint и т.п.). Каждый из этих снабженных префиксом On виртуальных методов является обработчиком соответствующего события по умолчанию. Если вы переопределите любой из этих виртуальных членов, то получите возможность выполнять любую необходимую пред- и пост-обработку события перед (или после) вызова реализации родительского класса:

```
public partial class MainWindow : Form
{
  protected override void OnMouseDown(MouseEventArgs e)
  {
```

```
    // Добавить специальный код для события MouseDown.
    // Затем вызвать родительскую реализацию.
    base.OnMouseDown(e);
  }
}
```

Хотя это может быть полезно при некоторых условиях (особенно если вы строите специальный элемент управления, который наследуется от стандартного элемента), часто вы будете обрабатывать события, используя стандартный синтаксис событий C# (фактически, это является поведением по умолчанию дизайнеров Visual Studio). Когда вы обрабатываете события подобным образом, каркас вызывает специальный обработчик события сразу после завершения родительской реализации:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    MouseDown += new MouseEventHandler(MainWindow_MouseDown);
  }

  private void MainWindow_MouseDown(object sender, MouseEventArgs e)
  {
    // Добавить код для события MouseDown.
  }
}
```

Помимо этих методов `OnXXX()` полезно знать несколько других методов.

- `Hide()`. Скрывает элемент управления и устанавливает свойство `Visible` в `false`.
- `Show()`. Отображает элемент управления и устанавливает свойство `Visible` в `true`.
- `Invalidate()`. Заставляет элемент управления перерисовывать себя, посылая событие `Paint`.

Класс `Control` определяет дополнительные свойства, методы и события помимо рассмотренного выше подмножества. Теперь вы должны иметь достаточное представление об общей функциональности этого базового класса. Давайте посмотрим на него в действии.

## Упражнения с классом `Control`

Чтобы проиллюстрировать пользу от некоторых членов класса `Control`, давайте построим новую форму, которая сможет обрабатывать следующие события:

- реагировать на события `MouseMove` и `MouseDown`;
- захватывать и обрабатывать клавиатурный ввод через событие `KeyUp`.

Для начала создайте новый класс-наследник `Form`. В конструкторе по умолчанию используются различные унаследованные свойства для установки начального внешнего вида и поведения. Обратите внимание, что теперь мы добавляем ссылку `using` на пространство имен `System.Drawing` для получения доступа к структуре `Color`:

```
using System;
using System.Windows.Forms;
using System.Drawing;
namespace MyWindowsApp
{
  public class MainWindow : Form
  {
```

```
    public MainWindow()
    {
      // Использовать унаследованные свойства для настройки
      // базового пользовательского интерфейса.
      Text = "My Fantastic Form";
      Height = 300;
      Width = 500;
      BackColor = Color.LemonChiffon;
      Cursor = Cursors.Hand;
    }
  }
  public static class Program
  {
    static void Main(string[] args)
    {
      Application.Run(new MainWindow());
    }
  }
}
```

Скомпилируйте приложение:

```
csc /target:winexe *.cs
```

## Реакция на событие MouseMove

Затем нужно обработать событие MouseMove. Целью будет отображение текущего положения (*x*, *y*) внутри области заголовка формы. Все события мыши (MouseMove, MouseUp, и т.п.) работают в сочетании с делегатом MouseEventHandler, который может вызывать любой метод, соответствующий следующей сигнатуре:

```
void MyMouseHandler(object sender, MouseEventArgs e);
```

Входная структура MouseEventArgs расширяет базовый класс EventArgs, добавляя множество членов, имеющих отношение к обработке активности мыши (см. табл. 2.6).

**Таблица 2.6. Свойства типа MouseEventArgs**

| Свойство | Назначение |
|---|---|
| Button | Определяет, какая кнопка мыши нажата, как определено в перечислении MouseButtons. |
| Clicks | Получает количество нажатий и отпусканий кнопки мыши. |
| Delta | Получает счетчик (со знаком) щелчков поворота колесика мыши. |
| X | Получает координату *x* щелчка мыши. |
| Y | Получает координату *y* щелчка мыши. |

Ниже приведен обновленный класс MainForm, обрабатывающий событие MouseMove:

```
public class MainForm : Form
{
  public MainForm()
  {
  ...
    // Обработать событие MouseMove
    MouseMove += new MouseEventHandler(MainForm_MouseMove);
  }
```

```
  // Обработчик события MouseMove.
  public void MainForm_MouseMove(object sender, MouseEventArgs e)
  {
    Text = string.Format("Current Pos: ({0}, {1})", e.X, e.Y);
  }
}
```

Запустив приложение и перемещая курсор мыши над окном, вы обнаружите позицию, отображаемую в области заголовка типа MainWindow (рис. 2.4).

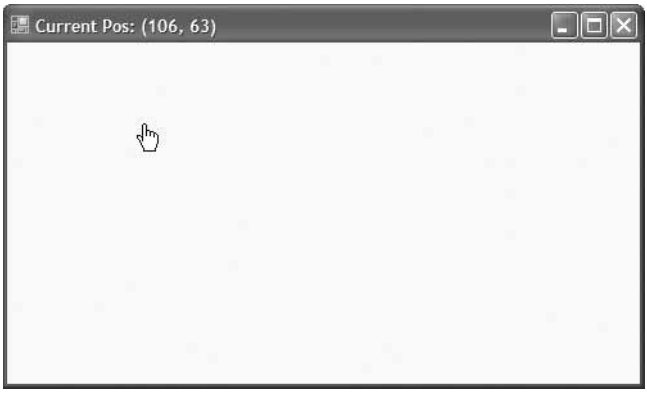


**Рис. 2.4.** Отслеживание движения курсора мыши

## Определение кнопки мыши, которой выполнен щелчок

Другая деталь, касающаяся активности мыши — определение нажатой кнопки при возникновении событий MouseUp, MouseDown, MouseClick или MouseDoubleClick. Когда вы хотите определить точно, какой кнопкой мыши была совершен щелчок (левой, правой или средней), необходимо проверить свойство Button класса MouseEventArgs. Значение свойства Button ограничено соответствующим перечислением MouseButtons. Предположим, что конструктор по умолчанию для обработки события MouseUp изменен следующим образом:

```
public MainWindow()
{
...
  // Обработать событие MouseUp.
  MouseUp += new MouseEventHandler(MainForm_MouseUp);
}
```

Следующий обработчик события MouseUp показывает, какая кнопка мыши была нажата внутри окна сообщения:

```
private void MainWindow_MouseUp (object sender, MouseEventArgs e)
{
  // Какой кнопкой мыши совершен щелчок?
  if(e.Button == MouseButtons.Left)
    MessageBox.Show("Left click!");
  if(e.Button == MouseButtons.Right)
    MessageBox.Show("Right click!");
  if (e.Button == MouseButtons.Middle)
    MessageBox.Show("Middle click!");
}
```

## Обработка событий клавиатуры

Обработка клавиатурного ввода почти идентична реакции на активность мыши. События KeyUp и KeyDown работают в сочетании с делегатом KeyEventHandler, который может указывать на любой метод, принимающий объект в первом параметре и KeyEventArgs во втором:

```
void MyKeyboardHandler(object sender, KeyEventArgs e);
```

Некоторые из наиболее интересных свойств, поддерживаемых KeyEventArgs, перечислены в табл. 2.7.

**Таблица 2.7. Свойства типа KeyEventArgs**

| Свойство | Назначение |
|---|---|
| Alt | Получает значение, указывающее на то, была ли нажата клавиша <Alt>. |
| Control | Получает значение, указывающее на то, была ли нажата клавиша <Ctrl>. |
| Handled | Получает или устанавливает значение, указывающее на то, было ли событие полностью обработано вашим обработчиком. |
| KeyCode | Получает код клавиши для событий KeyDown или KeyUp. |
| Modifiers | Указывает, какие модифицирующие клавиши (<Ctrl>, <Shift> и/или <Alt>) были нажаты. |
| Shift | Получает значение, указывающее на то, нажата ли клавиша <Shift>. |

Дополните класс MainForm обработкой события KeyUp. После этого отобразите имя клавиши, которая была нажата, внутри окна сообщения, используя свойство KeyCode.

```
public class MainForm : Form
{
  public MainForm()
  {
  ...
    // Слушать событие KeyUp.
    KeyUp += new KeyEventHandler(MainForm_KeyUp);
  }

  private void MainForm_KeyUp (object sender, KeyEventArgs e)
  {
    MessageBox.Show(e.KeyCode.ToString(), "Key Pressed!");
  }
}
```

Скомпилируйте и запустите программу. Теперь вы должны иметь возможность не только определить, какая кнопка мыши была нажата, но также и какая клавиша клавиатуры.

На этом обзор основной функциональности базового класса Control завершен. Давайте перейдем к изучению роли класса Form.

---

**Исходный код.** Проект ControlBehaviors включен в подкаталог Bonus Chapter 2.

# Функциональность класса Form

Класс Form обычно (но не обязательно) является непосредственным базовым классов специальных типов Form. В дополнение к огромному набору членов, унаследованных от классов Control, ScrollableControl и ContainerControl, тип Form добавляет дополнительную функциональность — в частности, главным окнам, дочерним окнам MDI и диалоговым окнам. Давайте начнем с основных свойств, перечисленных в табл. 2.8.

**Таблица 2.8. Свойства типа Form**

| Свойства | Назначение |
|---|---|
| AcceptButton | Получает или устанавливает кнопку формы, которая нажимается по нажатию клавиши <Enter>. |
| ActiveMDIChild<br>IsMDIChild<br>IsMDIContainer | Используются в контексте приложения MDI. |
| CancelButton | Получает или устанавливает кнопку, которая нажимается по нажатию пользователем клавиши <Esc>. |
| ControlBox | Получает или устанавливает значение, указывающее на то, имеет ли форма кнопку управляющего меню. |
| FormBorderStyle | Получает или устанавливает стиль рамки формы. Используется в сочетании с перечислением FormBorderStyle. |
| Menu | Получает или устанавливает меню, прикрепленное к форме. |
| MaximizeBox<br>MinimizeBox | Используются для определения того, имеет ли данная форма кнопки разворачивания и сворачивания. |
| ShowInTaskbar | Определяет, будет ли данная форма видна в панели задач Windows. |
| StartPosition | Получает или устанавливает начальную позицию формы во время выполнения, как указано перечислением FormStartPosition. |
| WindowState | Конфигурирует отображение формы при запуске. Используется в сочетании с перечислением FormWindowState. |

В дополнение к многочисленным обработчикам событий по умолчанию с префиксом On, в табл. 2.9 приведен список основных методов, определенных типом Form.

**Таблица 2.9. Методы типа Form**

| Метод | Назначение |
|---|---|
| Activate() | Активизирует данную форму и передает ей фокус. |
| Close() | Закрывает форму. |
| CenterToScren() | Помещает форму в центр экрана. |
| LayoutMDI() | Располагает каждое дочернее окно (как указано в перечислении LayoutMDI) внутри родительской формы. |
| ShowDialog() | Отображает форму как модальное диалоговое окно. |

И, наконец, в классе Form определен ряд событий, многие из которых инициируются на протяжении жизненного цикла формы. В табл. 2.10 перечислены важнейшие из них.

**Таблица 2.10. Избранные события типа Form**

| Событие | Назначение |
|---|---|
| Activated | Происходит при каждой *активизации* формы, т.е. когда форма получает фокус на рабочем столе. |
| Closed, Closing | Используется для определения момента, когда форма собирается закрываться и когда закрывается. |
| Deactivate | Происходит при каждой *деактивизации* формы, т.е. когда форма теряет текущий фокус на рабочем столе. |
| Load | Происходит после выделения памяти форме, но перед тем, как она становится видимой на экране. |
| MDIChildActive | Посылается при активизации дочернего окна. |

## Жизненный цикл типа Form

Если вам приходилось программировать пользовательские интерфейсы, используя наборы инструментов GUI типа Java Swing, Mac OS X Cocoa, или чистый Win32 API, вы знаете, что "оконные типы" имеют множество событий, которые происходят на протяжении времени их жизни. То же самое касается Windows Forms. Как вы уже видели, существование формы начинается с вызова конструктора типа перед тем, как она передается методу Application.Run().

Как только объект размещен в управляемой куче, каркас инициирует событие Load. Внутри обработчика события Load можно настроить внешний вид и поведение Form, подготовить любые вложенные дочерние элементы управления (такие как ListBox, TreeView и т.д.) либо просто выделить ресурсы, используемые в работе Form (соединения с базой данных, прокси на удаленные объекты и т.п.).

Как только событие Load инициировано, следующее за ним — Activated. Это событие происходит, когда форма получает фокус как активное окно на рабочем столе. Логическое дополнение события Activated — Deactivate, которое инициируется, когда форма теряет фокус как активное окно. Как вы можете догадаться, события Activated и Deactivate инициируются много раз на протяжении времени жизни экземпляра типа Form, по мере того, как пользователь перемещается между активными приложениями.

Когда пользователь пытается закрыть выбранную форму, происходят два события: Closing и Closed. Событие Closing инициируется первым и является идеальным местом для того, чтобы выдать пользователю самое ненавистное (но полезное) сообщение "Вы уверены, что хотите закрыть приложение?". Этот шаг подтверждения полезен тем, что дает пользователю шанс сохранить свои данные перед прекращением работы программы.

Событие Closing работает в сочетании с делегатом CancelEventHandler, определенным в пространстве имен System.ComponentModel. Если вы установите свойство CancelEventArgs.Cancel в true, то предотвратите закрытие окна и вернетесь к нормальной работе. Если же установите CancelEventArgs.Cancel в false, то будет инициировано событие Closed, и приложение Windows Forms завершится, AppDomain будет выгружен из памяти и процесс прерван.

Чтобы закрепить последовательность событий, происходящих на протяжении времени жизни формы, предположим, что имеется новый файл MainWindow.cs, устанавливающий обработку событий Load, Activated, Deactivate, Closing и Closed внутри конструктора класса (не забудьте добавить директиву using для пространства имен System.ComponentModel, чтобы получить определение CancelEventArgs):

```
public MainForm()
{
  // Обработать различные события жизненного цикла.
  Closing += new CancelEventHandler(MainForm_Closing);
  Load += new EventHandler(MainForm_Load);
  Closed += new EventHandler(MainForm_Closed);
  Activated += new EventHandler(MainForm_Activated);
  Deactivate += new EventHandler(MainForm_Deactivate);
}
```

Внутри обработчиков событий Load, Closed, Activated и Deactivate вы обновляете значение новой строковой переменной-члена уровня Form (по имени lifeTimeInfo) простым сообщением, которое отображает имя перехваченного события. К тому же обратите внимание, что внутри обработчика события Closed вы отображаете значение этой строки в окне сообщения:

```
private void MainForm_Load(object sender, System.EventArgs e)
{ lifeTimeInfo += "Load event\n"; }
private void MainForm_Activated(object sender, System.EventArgs e)
{ lifeTimeInfo += "Activate event\n"; }
private void MainForm_Deactivate(object sender, System.EventArgs e)
{ lifeTimeInfo += "Deactivate event\n"; }
private void MainForm_Closed(object sender, System.EventArgs e)
{
   lifeTimeInfo += "Closed event\n";
   MessageBox.Show(lifeTimeInfo);
}
```

Внутри обработчика события Closing вы предложите пользователю подтвердить его намерение завершить работу приложения, используя входящий CancelEventArgs:

```
private void MainForm_Closing(object sender, CancelEventArgs e)
{
   DialogResult dr = MessageBox.Show("Do you REALLY want to close this app?",
      "Closing event!", MessageBoxButtons.YesNo);
   if (dr == DialogResult.No)
      e.Cancel = true;
   else
      e.Cancel = false;
}
```

Обратите внимание, что метод MessageBox.Show() возвращает тип DialogResult, содержащий значение, указывающее на то, какая кнопка была нажата пользователем в окне сообщения (Yes или No). Затем перекомпилируйте код в командной строке:

```
csc /target:winexe *.cs
```

Теперь запустите приложение и несколько раз переключите фокус с формы и обратно (чтобы инициировать события Activated и Deactivate). Как только вы в конечном итоге остановите приложение, то увидите окно сообщения вроде приведенного на рис. 2.5.


**Рис. 2.5.** Последовательность событий жизненного цикла типа-наследника Form

Но наиболее интересные аспекты типа Form касаются его способности создавать и размещать в себе системы меню, панели инструментов и линейки состояния. Хотя необходимый для этого код не сложен, вы обрадуетесь, узнав, что Visual Studio включает несколько графических дизайнеров, которые позаботятся о генерации рутинного кода.

Учитывая это, давайте пока попрощаемся с компилятором командной строки и переключим внимание на процесс построения приложений Windows Forms с применением Visual Studio.

**Исходный код.** Проект FormLifeTime находится в подкаталоге Bonus Chapter 2.

# Построение приложений Windows в среде Visual Studio

Visual Studio включает специфический тип проектов, предназначенный для создания приложений Windows Forms. Когда вы выбираете тип проекта Windows Application, то не только получаете объект приложения с правильным методом Main(), но также имеете начальный тип-наследник Form. Еще лучше то, что IDE-среда предлагает несколько графических дизайнеров, которые превращают процесс построения пользовательского интерфейса в игру. Просто чтобы почувствовать вкус, создайте новое рабочее пространство проекта Windows Application (рис. 2.6). Пока еще вы не будете строить работающий пример, так что назовите проект произвольным образом.

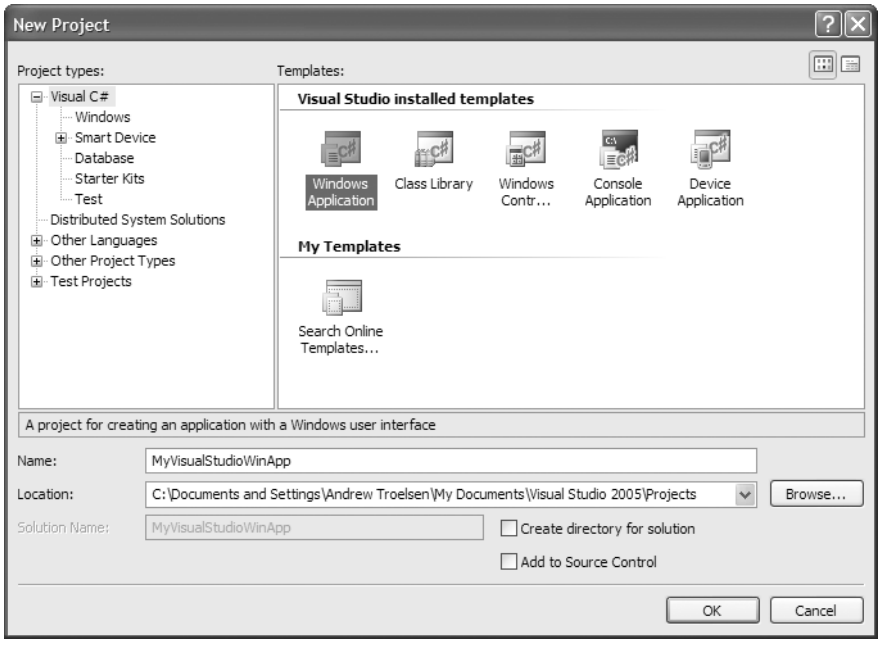


**Рис. 2.6.** Проект Windows Application в среде Visual Studio

Как только проект загружен, вы сразу увидите дизайнер форм, который позволит строить пользовательский интерфейс посредством перетаскивания элементов управления/компонентов из панели инструментов Toolbox (рис. 2.7) и сконфигурируйте их свойства и события, используя окно Properties (рис. 2.8).

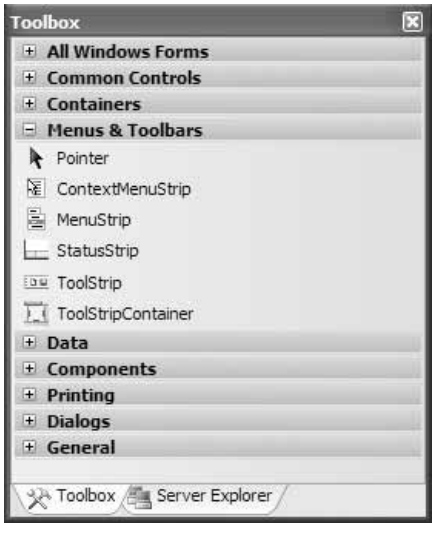


**Рис. 2.7.** Панель инструментов Toolbox

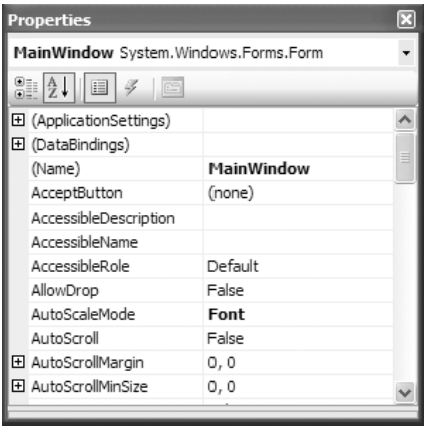


**Рис. 2.8.** Окно Properties

Как видите, панель инструментов Toolbox группирует элементы управления по различным категориям. Хотя большинство из них самоочевидны (например, Printing (Печать) содержит элементы управления печатью, Menus & Toolbars (Меню и панели инструментов) — рекомендуемые элементы меню и панелей инструментов, и т.д.), некоторые категории нуждаются в специальном упоминании.

- Common Controls (Общие элементы управления). Члены этой категории считаются «рекомендованным набором» распространенных элементов пользовательского интерфейса.
- All Windows Forms (Все элементы управления Windows Forms). Здесь вы найдете полный набор элементов управления Windows Forms, включая различные элементы .NET 1.x, которые считаются устаревшими.

Второй пункт требует дополнительных пояснений. Если вы работали с Windows Forms в .NET 1.x, имейте в виду, что многие из ваших старых друзей (такие как элемент управления `DataGrid`), помещены в категорию All Windows Forms. Более того, распространенные элементы пользовательского интерфейса, которые применялись в .NET 1.x (вроде `MainMenu`, `ToolBar` и `StatusBar`) по умолчанию в Toolbox не отображаются.

## Включение устаревших элементов управления

Первая порция хороших новостей состоит в том, что эти (устаревшие) элементы интерфейса все еще доступны в .NET 2.0. Вторая порция хороших новостей в том, что если вы хотите программировать с их применением, можете вернуть их в панель Toolbox, выполнив щелчок правой кнопкой мыши где-нибудь на поверхности Toolbox и выбрав в контекстном меню пункт Choose Items (Выбрать элементы). В результирующем диалоговом окне отметьте все интересующие элементы (рис. 2.9).

---

**На заметку!** На первый взгляд может показаться, что есть избыточные позиции по некоторым элементам управления (таким как `ToolBar`). На самом деле каждое их появление уникально, потому что элементы управления могут иметь версии (1.0 или 2.0) и/или быть членами .NET Compact Framework. Проверяйте путь каталога и выбирайте правильные элементы.

---

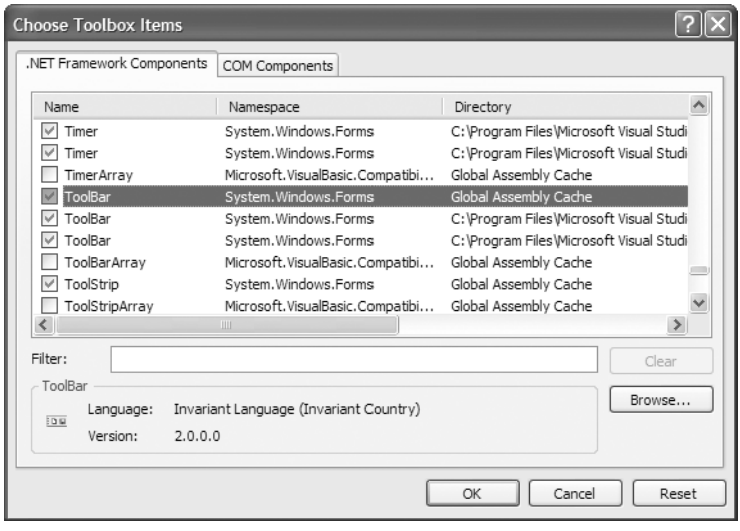


**Рис. 2.9.** Добавление дополнительных элементов управления в панель Toolbox

Наверняка вы недоумеваете, почему так много старых знакомых из числа элементов управления скрыты от глаз? Причина в том, что в .NET 2.0 появился набор новых меню, панелей инструментов и линеек состояния, применение которых предпочтительно. Например, вместо унаследованного `MainMenu` для построения меню, можно использовать элемент `MenuStrip`, который предлагает массу полезных свойств в дополнение к функциональности, имеющейся в `MainMenu`.

---

**На заметку!** В этой главе предпочтение отдается использованию нового рекомендованного набора элементов пользовательского интерфейса. Если вы хотите работать с унаследованными типами `MainMenu`, `StatusBar` или `Toolbar`, загляните в документацию по .NET Framework 2.0 SDK.

---

## Внутренняя структура проекта Windows Forms

Каждый класс формы в проекте Windows Forms состоит из двух связанных файлов C#, которые можно увидеть в Solution Explorer (рис. 2.10).

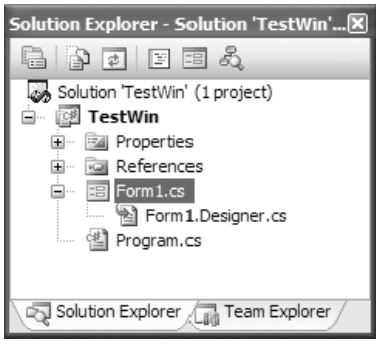


**Рис. 2.10.** Каждый класс `Form` состоит из двух файлов `*.cs`

Выполните щелчок правой кнопкой мыши на пиктограмме Form1.cs и выберите в контекстном меню пункт View Code (Просмотреть код). Здесь вы увидите частичный класс, содержащий все обработчики событий Form, конструкторы, переопределения и все члены, написанные вами (обратите внимание, что начальный класс Forms1 переименован в MainWindow с помощью рефакторинга Rename):

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

Конструктор по умолчанию формы выполняет вызов метода по имени InitializeComponent(), который определен в связанном файле *.Designer.cs. Этот метод поддерживается средой Visual Studio и содержит весь код, представляющий модификации, выполненные в дизайнере.

Чтобы проиллюстрировать, переключитесь обратно в дизайнер форм и найдите свойство Text в окне Properties. Измените это значение на что-то вроде My Test Windows. Теперь откройте файл Form1.Designer.cs и обратите внимание на соответствующие изменения в InitializeComponent():

```
private void InitializeComponent()
{
...
    this.Text = "My Test Window";
}
```

В дополнение к поддержке InitializeComponent(), файл *.Designer.cs определяет переменные-члены, которые представляют каждый элемент управления, добавленный в дизайнере. Опять-таки, для иллюстрации перетащите элемент управления Button в дизайнер форм. В окне Properties переименуйте переменную-член с button1 на btnTestButton через свойство Name.

---

**На заметку!** Переименовывать элементы управления, добавляемые к форме в дизайнере перед обработкой их событий — всегда хорошая идея. Если вы этого не сделаете, то, скорее всего, столкнетесь с множеством непонятных обработчиков событий, вроде button27_Click, учитывая, что имена по умолчанию образуются просто добавлением числового суффикса к имени переменной.

---

## Обработка событий во время проектирования

Обратите внимание, что окно Properties содержит кнопку с изображением молнии. Хотя вы всегда вольны обрабатывать события уровня формы написанием нужной логики вручную (как делалось в предыдущих примерах), эта кнопка позволяет визуально обработать событие заданного элемента управления. Просто выберите необходимый элемент управления в раскрывающемся списке (в верхней части окна Properties), найдите интересующее событие и наберите имя метода, который будет использован в качестве обработчика события (или просто выполните двойной щелчок на событии, чтобы сгенерировать имя по умолчанию в виде ИмяЭлемента _ ИмяСобытия).

Предполагая, что вы обработали событие Click элемента управления Button1, вы обнаружите в файле Form1.cs следующий обработчик:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    InitializeComponent();
  }
  private void btnButtonTest_Click(object sender, EventArgs e)
  {
  }
}
```

Кроме того, файл Form1.Designer.cs будет содержать необходимую инфраструктуру и объявление переменной-члена:

```
partial class MainWindow
{
...
  private void InitializeComponent()
  {
  ...
    this.btnButtonTest.Click +=
        new System.EventHandler(this.btnButtonTest_Click);
  }
  private System.Windows.Forms.Button btnButtonTest;
}
```

**На заметку!** Каждый элемент управления имеет *событие по умолчанию*, которое указывает событие, обработчик которого будет открыт при двойном щелчке на элементе в дизайнере форм. Например, событием по умолчанию для Form является Load, и если выполнить двойной щелчок в любом месте типа Form, среда IDE автоматически сгенерирует код обработки этого события.

## Класс Program

Помимо файлов, связанных с формой, приложение Windows в Visual Studio определяет второй класс, который представляет объект приложения (т.е. тип, определяющий метод Main()). Обратите внимание, что метод Main() сконфигурирован для вызова Application.EnableVisualStyles(), а также Application.Run():

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

**На заметку!** Атрибут [STAThread] инструктирует CLR о том, что любые унаследованные объекты COM (включая элементы управления ActiveX) следует размещать в однопоточном апартаменте (single-threaded appartment — STA). Если вы имеете опыт работы с COM, то можете вспомнить, что STA используется для обеспечения доступа к типу COM в синхронной (и потому безопасной к потокам) манере.

## Автоматически ссылаемые сборки

И, наконец, если вы заглянете в Solution Explorer, то заметите, что проект Windows Forms автоматически ссылается на ряд сборок, включая System.Windows.Forms.dll и System.Drawing.dll. Подробности относительно этого будут изложены в следующей главе.

# Работа с элементами MenuStrip и ContextMenuStrip

В .NET 2.0 рекомендуемым элементом управления для построения систем меню является MenuStrip. Этот элемент позволяет строить "нормальные" пункты меню вроде File⇨Exit (Файл⇨Выход); кроме того, его можно конфигурировать для расположения в меню любого количества других элементов управления. Вот некоторые распространенные элементы пользовательского интерфейса, которые могут содержаться внутри MenuStrip:

- ToolStripMenuItem — традиционный пункт меню;
- ToolStripComboBox — встроенный раскрывающийся список;
- ToolStripSeparator — простая линия, разделяющая содержимое;
- ToolStripTextBox — встроенное текстовое поле.

Говоря языком программиста, элемент управления MenuStrip содержит строго типизированную коллекцию, именуемую ToolStripItemCollection. Подобно другим типам коллекций, этот объект поддерживает такие члены, как Add(), AddRange(), Remove() и свойство Count. Хотя эта коллекция обычно наполняется опосредованно с использованием различных инструментов дизайна, ее содержимым можно манипулировать вручную.

Чтобы проиллюстрировать процесс работы с элементом управления MenuStrip, создайте новое приложение Windows Forms по имени MenuStripApp. С помощью дизайнера форм поместите элемент управления MenuStrip по имени MainMenuStrip на форму. После этого файл *.Designer.cs будет дополнен новой переменной-членом типа MenuStrip:

```
private System.Windows.Forms.MenuStrip mainMenuStrip;
```

В дизайнере форм можно очень тонко настроить MenuStrip. Например, в крайнем верхнем левом углу элемента управления находится маленькая пиктограмма со стрелкой. Выбрав эту пиктограмму, вы получите контекстно-чувствительный "встроенный редактор", показанный на рис. 2.11.

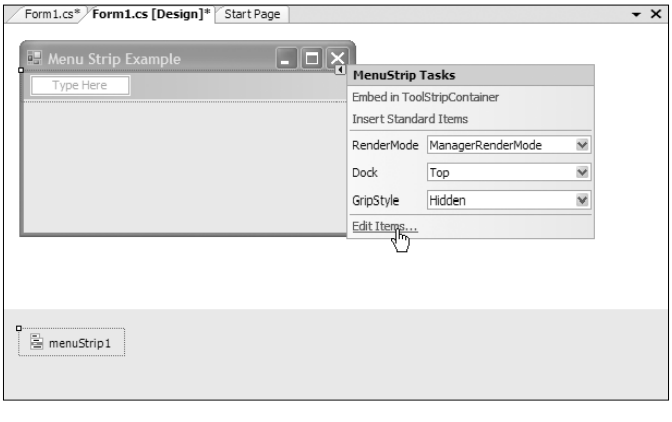


**Рис. 2.11.** Встроенный редактор MenuStrip

Многие элементы управления Windows Forms поддерживают такие контекстно-зависимые встроенные редакторы. Что касается MenuStrip, то его редактор позволяет быстро выполнять следующие действия:

- вставлять "стандартную" систему меню (File (Файл), Save (Сохранить), Tools (Сервис), Help (Справка) и т.д.), используя ссылку Insert Standard Items (Вставить стандартные элементы);
- изменять поведение группировки и стыковки элемента MenuStrip;
- редактировать элемент в MenuStrip (это просто ярлык для выбора определенного элемента в окне Properties).

В данном примере мы будем игнорировать опции встроенного редактора и сосредоточимся на дизайне системы меню. Для начала выберите элемент MenuStrip в дизайнере и определите стандартное меню File⇨Exit, введя имена внутри приглашений Type Here (Набирайте здесь), как показано на рис. 2.12.

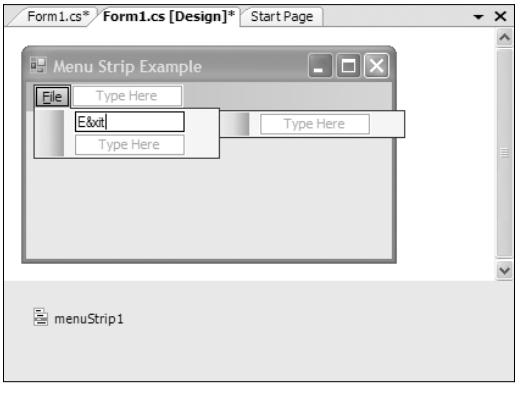


**Рис. 2.12.** Проектирование системы меню

**На заметку!** Как вы, возможно, знаете, когда перед буквой в пункте меню используется знак амперсанда (&), это определяет горячую клавишу для данного элемента. В рассматриваемом примере был создан &File⇨E&xit, следовательно, пользователь сможет активизировать пункт меню Exit, нажав <Alt+f>, а затем — <x>.

Каждый пункт меню, который вы вводите в дизайнере, представлен элементом типа класса ToolStripMenuItem. Открыв файл *.Designer.cs, вы найдете там две новых переменных-члена для каждого элемента меню:

```
partial class MainWindow
{
...
    private System.Windows.Forms.MenuStrip mainMenuStrip;
    private System.Windows.Forms.ToolStripMenuItem fileToolStripMenuItem;
    private System.Windows.Forms.ToolStripMenuItem exitToolStripMenuItem;
}
```

Когда вы используете редактор меню, метод InitializeComponent() соответствующим образом обновляется. Обратите внимание, что внутренняя коллекция ToolStripItemCollection экземпляра MenuStrip обновляется добавлением нового пункта меню высшего уровня (fileToolStripMenuItem). Аналогичным образом переменная fileToolStripMenuItem обновляется вставкой переменной exitToolStrip MenuItem в ее коллекцию ToolStripItemCollection через свойство DropDownItems:

```
private void InitializeComponent()
{
...
  //
  // menuStrip1
  //
  this.menuStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
    this.fileToolStripMenuItem});
...
  //
  // fileToolStripMenuItem
  //
  this.fileToolStripMenuItem.DropDownItems.AddRange(new
    System.Windows.Forms.ToolStripItem[] { this.exitToolStripMenuItem});
...
  //
  // MainWindow
  //
  this.Controls.Add(this.menuStrip1);
}
```

И последнее: обратите внимание, что элемент управления MenuStrip вставлен в коллекцию элементов Form, Чтобы элемент управления был видимым во время выполнения, он должен быть членом этой коллекции. Для завершения начального кода этого примера, вернитесь в дизайнер и обработайте событие Click пункта меню Exit, используя кнопку событий окна Properties. Внутри сгенерированного обработчика события вызовите Application.Exit:

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

После этого можно скомпилировать и запустить программу. Удостоверьтесь, что можно завершить приложение, выбрав пункт меню File⇨Exit, а также нажав на клавиатуре <Alt+f> , затем <x>.

## Добавление текстового поля в MenuStrip

Теперь давайте создадим новое меню верхнего уровня по имени Set Background Color (Установить цвет фона). Подэлементом на этот раз будет не пункт меню, а ToolStripTextBox (рис. 2.13). Добавив новый элемент, переименуйте его в toolStripTextBoxColor в окне Properties.

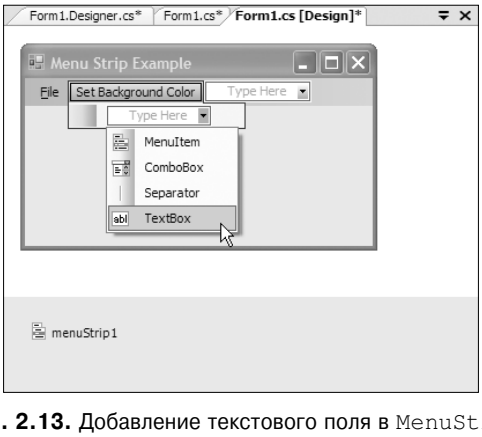


**Рис. 2.13.** Добавление текстового поля в MenuStrip

Нашей целью здесь будет позволить пользователю ввести название цвета (red, green, pink и т.д.), который будет использован для установки свойства BackColor формы. Для начала обработайте событие LostFocus новой переменной-члена ToolStripTextBox внутри конструктора формы (как и предполагалось, это событие инициируется, когда TextBox внутри ToolStrip перестает быть активным элементом интерфейса):

```
public MainWindow()
{
...
   toolStripTextBoxColor.LostFocus
      += new EventHandler(toolStripTextBoxColor_LostFocus);
}
```

Внутри сгенерированного обработчика события строковые данные, введенные в ToolStripTextBox, извлекаются через свойство Text и вызывается метод System.Drawing.Color.FromName(). Этот статический метод вернет тип Color на основе известного строкового значения. Чтобы учесть возможность ввода пользователем неизвестного цвета (или ошибочных данных), применяется логика try/catch:

```
void toolStripTextBoxColor_LostFocus(object sender, EventArgs e)
{
   try
   {
      BackColor = Color.FromName(toolStripTextBoxColor.Text);
   } catch { } // Ничего не делать, если пользователь ввел неверные данные.
}
```

Теперь снова попробуйте запустить приложение и ввести названия разных цветов. Вы должны увидеть изменение цвета фона вашей формы. Если хотите узнать, какие имена цветов допустимы, загляните в тип System.Drawing.Color, используя Object Browser или документацию по .NET Framework 2.0 SDK.

## Создание контекстного меню

Теперь давайте рассмотрим процесс построения контекстного всплывающего меню (доступного по щелчку правой кнопкой мыши). В .NET 1.1 тип ContextMenu был единственным классом для построения контекстных меню, но в .NET 2.0 предпочтительным типом для этой цели стал ContextMenuStrip. Подобно типу MenuStrip, ContextMenuStrip поддерживает ToolStripItemCollection для представления возможных подэлементов (таких как ToolStripMenuItem, ToolStripComboBox, ToolStripSeperator, ToolStripTextBox и т.д.).

Перетащите новый элемент управления ContextMenuStrip из панели Toolbox в дизайнер форм и переименуйте его в fontSizeContextStrip, используя окно Properties. Обратите внимание, что вы можете наполнить подэлементы графически, почти так же, как делали это с главным меню MenuStrip формы (замечательное отличие от метода, применявшегося в Visual Studio 2003). Для целей этого примера добавьте три элемента ToolStripMenuItem по имени Huge, Normal и Tiny (рис. 2.14).

Это контекстное меню будет использоваться, чтобы позволить пользователю выбирать размер визуализации сообщения в клиентской области формы. Чтобы облегчить задачу, создайте новый тип перечисления по имени TextFontSize в пространстве имен MenuStripApp и объявите новую переменную-член этого типа внутри типа Form (установите в TextFontSize.FontSizeNormal):

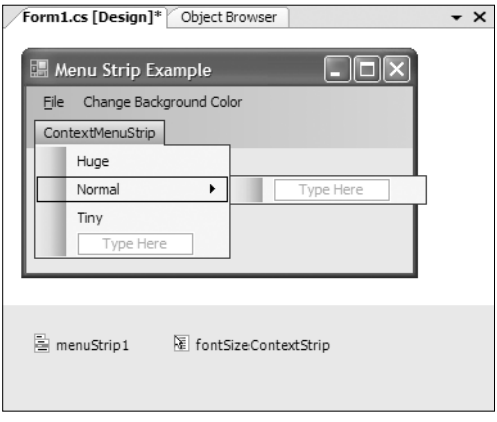


**Рис. 2.14.** Дизайн ContextMenuStrip

```
namespace MainForm
{
  // Вспомогательное перечисление для размеров шрифта.
  enum TextFontSize
  {
    FontSizeHuge = 30,
    FontSizeNormal = 20,
    FontSizeTiny = 8
  }
  public class MainForm : Form
  {
    // Текущий размер шрифта.
    private TextFontSize currFontSize
      = TextFontSize.FontSizeNormal;
  ...
  }
}
```

Следующий шаг состоит в обработке события формы Paint с использованием окна Properties. Событие Paint позволяет визуализировать графические данные (включая стилизованный текст) в клиентской области формы. Здесь вы будете рисовать текстовое сообщение, используя шрифт, указанный пользователем. Пока не беспокойтесь о деталях, а просто модифицируйте обработчик события Paint следующим образом:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.DrawString("Right click on me...",
          new Font("Times New Roman", (float)currFontSize),
          new SolidBrush(Color.Black), 50, 50);
}
```

И последнее: необходимо обработать событие Click для каждого из экземпляров ToolStripMenuItem, поддерживаемых ContextMenuStrip. Хотя можно было бы завести отдельный обработчик события Click для каждого, здесь вы специфицируете единственный обработчик, который будет вызван при щелчке на любом из трех ToolStripMenuItem. Используя окно Properties, специфицируйте имя обработчика события Click как ContextMenuItemSelection_Clicked для каждого из трех элементов ToolStripMenuItem и реализуйте этот метод следующим образом:

```
private void ContextMenuItemSelection_Clicked(object sender, EventArgs e)
{
  // Получить текущий ToolStripMenuItem, на котором был совершен щелчок.
  ToolStripMenuItem miClicked =
    (ToolStripMenuItem)sender;

  // Определить, на каком элементе меню был совершен щелчок, по его Name.
  if (miClicked.Name == "hugeToolStripMenuItem")
    currFontSize = TextFontSize.FontSizeHuge;
  if (miClicked.Name == "normalToolStripMenuItem")
    currFontSize = TextFontSize.FontSizeNormal;
  if (miClicked.Name == "tinyToolStripMenuItem")
    currFontSize = TextFontSize.FontSizeTiny;

  // Заставить форму перерисовать себя.
  Invalidate();
}
```

Обратите внимание, что с использованием аргумента `sender` можно определить имя переменной-члена `ToolStripMenuItem` для установки текущего размера шрифта. После этого вызов `Invalidate()` инициирует событие `Paint`, что вызовет запуск обработчика события `Paint`.

Последний шаг — информировать `Form` о том, какой `ContextMenuStrip` должен отображаться при щелчке правой кнопкой мыши в его клиентской области. Для этого воспользуйтесь окном Properties для установки свойства `ContextMenuStrip` равным имени контекстного меню. Сделав это, вы найдете следующую строку в `InitializeComponent()`:

```
this.ContextMenuStrip = this.fontSizeContextStrip;
```

---

**На заметку!** Имейте в виду, что *любому* элементу управления может быть назначено контекстное меню через свойство `ContextMenuStrip`. Например, можно было бы создать объект `Button` в диалоговом окне, который реагирует на определенное контекстное меню. Таким образом, меню будет отображаться, только если правая кнопка мыши была нажата, пока курсор мыши находится в пределах границ кнопки.

---

Теперь приложение позволяет изменять размер визуализированного текстового сообщения щелчком правой кнопкой мыши.

## Отметка пунктов меню

`ToolStripMenuItem` определяет множество членов, позволяющих отмечать, включать и скрывать определенный элемент меню. В табл. 2.11 дан список некоторых интересных свойств.

**Таблица 2.11. Члены типа `ToolStripMenuItem`**

| Член | Назначение |
|---|---|
| `Checked` | Получает или устанавливает значение, указывающее на наличие отметки рядом с текстом `ToolStripMenuItem`. |
| `CheckOnClick` | Получает или устанавливает значение, указывающее на то, должен ли `ToolStripMenuItem` автоматически становиться отмеченным/неотмеченным при щелчке. |
| `Enabled` | Получает или устанавливает значение, указывающее, доступен ли `ToolStripMenuItem`. |

Теперь давайте расширим предыдущее всплывающее меню, чтобы оно отображало отметку рядом с текущим выбранным пунктом. Установка метки на определенный пункт меню — совсем не сложная задача (просто установите свойство Checked равным true). Однако отслеживание того, какой именно элемент меню должен быть отмечен, требует некоторой дополнительной логики. Один из возможных подходов состоит в определении отдельной переменной-члена ToolStripMenuItem, которая представляет текущий помеченный элемент:

```
public class MainWindow : Form
{
...
  // Отмеченный элемент меню.
  private ToolStripMenuItem currentCheckedItem;
}
```

Напомним, что размером текста по умолчанию является TextFontSize.FontSizeNormal. С учетом этого, начальный элемент, который должен быть отмечен — это переменная-член normalToolStripMenuItem типа ToolStripMenuItem. Модифицируйте конструктор Form следующим образом:

```
public MainWindow()
{
  // Унаследованный метод для центрирования Form.
  CenterToScreen();
  InitializeComponent();
  // Отметить пункт меню Normal.
  currentCheckedItem = normalToolStripMenuItem;
  currentCheckedItem.Checked = true;
}
```

Теперь, имея возможность программно идентифицировать текущий отмеченный элемент, остается последний шаг — обновить обработчик события ContextMenuItem Selection_Clicked(), чтобы он снимал отметку с ранее отмеченного элемента и отмечал текущий объект ToolStripMenuItem в ответ на выбор пользователя:

```
private void ContextMenuItemSelection_Clicked(object sender, EventArgs e)
{
  // Снять отметку с текущего отмеченного пункта.
  currentCheckedItem.Checked = false;
...
  if (miClicked.Name == "hugeToolStripMenuItem")
  {
    currFontSize = TextFontSize.FontSizeHuge;
    currentCheckedItem = hugeToolStripMenuItem;
  }
  if (miClicked.Name == "normalToolStripMenuItem")
  {
    currFontSize = TextFontSize.FontSizeNormal;
    currentCheckedItem = normalToolStripMenuItem;
  }
  if (miClicked.Name == "tinyToolStripMenuItem")
  {
    currFontSize = TextFontSize.FontSizeTiny;
    currentCheckedItem = tinyToolStripMenuItem;
  }
  // Отметить элемент.
  currentCheckedItem.Checked = true;
...
}
```

На рис. 2.15 показан завершенный проект MenuStripApp в действии.

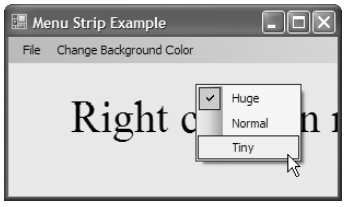


**Рис. 2.15.** Отметка и снятие отметки с `ToolStripMenuItem`

**Исходный код.** Проект `MenuStripApp` находится в подкаталоге `Bonus Chapter 2`.

# Работа с элементом StatusStrip

В дополнение к системе меню многие формы также поддерживают *линейку состояния* (status bar), которая обычно прикрепляется к нижней части `Form`. Линейка состояния может делиться на несколько "панелей", которые содержат некоторую текстовую или графическую информацию, наподобие строк справки меню, текущего времени или другой специфичной для приложения информации.

Хотя линейки состояния поддерживаются с самого появления платформы .NET (через тип `System.Windows.Forms.StatusBar`), начиная с .NET 2.0 простой `StatusBar` был вытеснен новым типом `StatusStrip`. Подобно простой линейке состояния, `StatusStrip` может состоять из любого количества панелей для хранения текстовых/графических данных с использованием типа `ToolStripStatusLabel`. Однако ленты состояния обладают способностью хранить дополнительные элементы, вроде перечисленных ниже.

- `ToolStripProgressBar`. Встроенный индикатор хода работ.
- `ToolStripDropDownButton`. Встроенная кнопка, отображающая при щелчке раскрывающийся список вариантов.
- `ToolStripSplitButton`. Подобна `ToolStripDropDownButton`, но элементы раскрывающегося списка отображаются, только когда пользователь щелкнет непосредственно в раскрывающейся области элемента. `ToolStripSplitButton` также обладает нормальным поведением кнопки и потому может поддерживать событие `Click`.

В данном примере мы построим окно `MainWindow`, которое поддерживает простые меню (File⇨Exit и Help⇨About), а также `StatusStrip`. Крайняя левая панель линейки состояния будет использоваться для отображения данных строки подсказки, связанной с текущим выбранным подпунктом меню (т.е. если пользователь выбирает меню Exit, в панели отобразится строка Exit the app (Завершить приложение)).

Крайняя правая панель отобразит одну или две динамически создаваемых строки, которые будут показывать текущее время или текущую дату. И, наконец, средняя панель отобразит кнопку `ToolStripDropDownButton`, которая позволит пользователю переключать отображение даты/времени (с пиктограммой при запуске). На рис. 2.16 показано приложение в готовом виде.

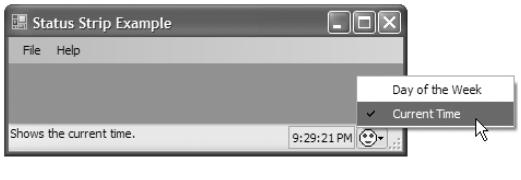


**Рис. 2.16.** Приложение `StatusStripApp`

## Дизайн системы меню

Для начала создайте проект приложения Windows Forms по имени StatusStripApp. Поместите элемент управления MenuStrip в дизайнер форм и создайте два пункта меню (File⇨Exit и Help⇨About). Обработайте события Click и MouseHover для каждого подэлемента с помощью окна Properties. Реализация обработчика событий Click пункта File⇨Exit просто останавливает приложение, а обработчик Click пункта Help⇨About отображает дружественный MessageBox:

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

private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{ MessageBox.Show("My StatusStripApp!"); }
```

Чуть позже вы измените обработчик событий MouseHover, чтобы он отображал корректную подсказку в левой панели StatusStrip, а пока оставьте его пустым.

## Дизайн StatusStrip

Затем поместите элемент управления StatusStrip в дизайнер и переименуйте его на mainStatusStrip. Имейте в виду, что по умолчанию StatusStrip не содержит никаких панелей. Для добавления трех панелей можно использовать различные подходы.

- Написать код вручную без поддержки дизайнера (возможно, используя вспомогательный метод по имени CreateStatusStrip(), который будет вызван конструктором Form).
- Добавить элементы через диалоговое окно, активизируемое по ссылке Edit Items (Редактировать элементы) с использованием контекстно-чувствительного встроенного редактора StatusStrip (рис. 2.17).
- Добавить элементы друг за другом через новый раскрывающийся редактор, прикрепленный к StatusStrip (рис. 2.18).

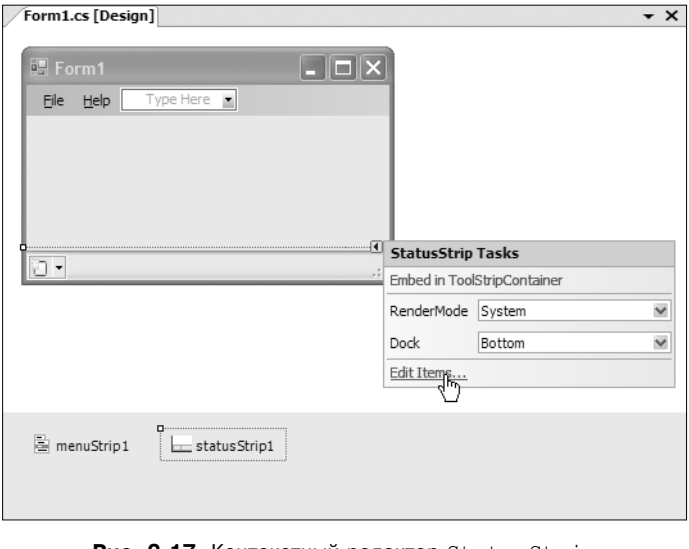


**Рис. 2.17.** Контекстный редактор StatusStrip

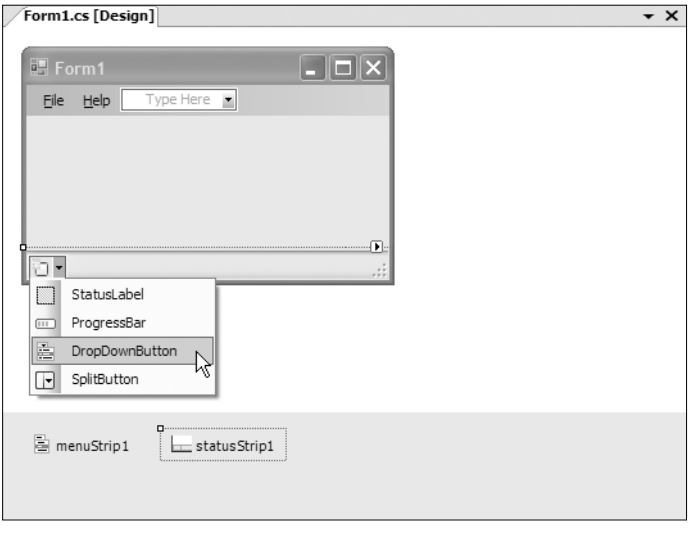


**Рис. 2.18.** Добавление элементов через новый встроенный раскрывающийся редактор элементов `StatusStrip`

В данном примере применяется новый раскрывающийся редактор элементов. Добавьте два новых элемента `ToolStripStatusLabel` по имени `toolStripStatusLabelMenuState` и `toolStripStatusLabelClock` и элемент `ToolStripDropDownButton` по имени `toolStripDropDownButtonDateTime`. Как и можно было ожидать, это приведет к добавлению новых переменных-членов в файл `*.Designer.cs` и соответствующему обновлению `InitializeComponent()`. Обратите внимание, что `StatusStrip` поддерживает внутреннюю коллекцию для хранения каждой панели:

```
partial class MainForm
{
  private void InitializeComponent()
  {
...
  //
  // mainStatusStrip
  //
  this.mainStatusStrip.Items.AddRange(new System.Windows.Forms.ToolStripItem[]
  {
     this.toolStripStatusLabelMenuState,
     this.toolStripStatusLabelClock,
     this.toolStripDropDownButtonDateTime});
...
  }
  private System.Windows.Forms.StatusStrip mainStatusStrip;
  private System.Windows.Forms.ToolStripStatusLabel
     toolStripStatusLabelMenuState;
  private System.Windows.Forms.ToolStripStatusLabel
     toolStripStatusLabelClock;
  private System.Windows.Forms.ToolStripDropDownButton
     toolStripDropDownButtonDateTime;
...
}
```

Теперь выберите в дизайнере ToolStripDropDownButton и добавьте два новых пункта меню по имени currentTimeToolStripMenuItem и dayoftheWeekToolStripMenuItem (рис. 2.19).

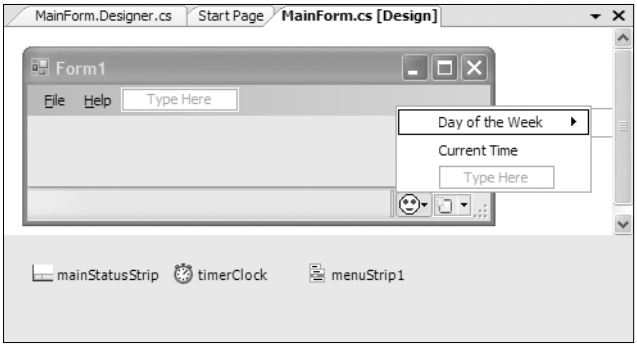


**Рис. 2.19.** Добавление пунктов меню к ToolStripDropDownButton

Для конфигурирования каждой панели, чтобы она выглядела, как показано на рис. 2.19, понадобится установить несколько свойств в окне Properties. В табл. 2.12 документированы необходимые свойства для установки и события для обработки каждого элемента в StatusStrip (разумеется, можно стилизовать панели дополнительными установками по своему усмотрению).

**Таблица 2.12. Конфигурация панелей StatusStrip**

| Переменная-член панели | Свойства для установки | События для обработки |
|---|---|---|
| toolStripStatusLabelMenuState | Spring = true<br>Text = (empty)<br>TextAlign = TopLeft | Нет |
| toolStripStatusLabelClock | BorderSides = All<br>Text = (empty) | Нет |
| toolStripDropDownButtonDateTime | Image = (see text that follows) | Нет |
| dayoftheWeekToolStripMenuItem | Text = "Day of the Week" | MouseHover<br>Click |
| currentTimeToolStripMenuItem | Text = "Current Time" | MouseHover<br>Click |

Свойство Image члена toolStripDropDownButtonDateTime может быть установлено в любой существующий на машине файл изображения (конечно, очень большие изображения будут сильно искажены). Для данного примера можно воспользоваться файлом happyDude.bmp из загружаемого кода для этой главы.

Итак, на дизайн графического интерфейса готов. Прежде чем вы реализуете остальные обработчики меню, следует ознакомиться с назначением компонента Timer.

## Работа с типом Timer

Напомним, что вторая панель должна отображать текущее время или текущую дату на основе предпочтений пользователя. Первый шаг для реализации этой цели дизайна

состоит в добавлении переменной-члена Timer к нашему классу Form. Класс Timer — это компонент, который вызывает некоторый метод (специфицированный с использованием события Tick) с заданным интервалом (специфицированным свойством Interval).

Перетащите компонент Timer на поверхность дизайнера форм и переименуйте его в timerDateTimeUpdate. Используя окно Properties, установите свойство Interval равным 1000 (значение в миллисекундах), а свойство Enabled — в true. И, наконец, обработайте событие Tick. Перед реализацией обработчика события Tick определите новый тип перечисления в проекте, именуемый DateTimeFormat. Это перечисление будет использовано для определения того, должен второй ToolStripStatusLabel отображать текущее время или текущий день недели:

```
enum DateTimeFormat
{
   ShowClock,
   ShowDay
}
```

Имея это перечисление, модифицируйте класс MainWindow следующим образом:

```
public partial class MainWindow : Form
{
  // Какой формат отображать?
  DateTimeFormat dtFormat = DateTimeFormat.ShowClock;
  ...
  private void timerDateTimeUpdate_Tick(object sender, EventArgs e)
  {
    string panelInfo = "";

    // Создать текущий формат.
    if (dtFormat == DateTimeFormat.ShowClock)
      panelInfo = DateTime.Now.ToLongTimeString();
    else
      panelInfo = DateTime.Now.ToLongDateString();

    // Установить текст панели.
    toolStripStatusLabelClock.Text = panelInfo;
  }
}
```

Обратите внимание, что обработчик событий Timer использует тип DataTime. Здесь вы просто определяете текущее системное время и дату с помощью свойства Now и применяете его для установки свойства Text переменной-члена toolStripStatusLabelClock.

## Переключение отображения

К этому моменту обработчик события Tick должен отображать текущее время внутри панели toolStripStatusLabelClock, учитывая, что значение переменной-члена DateTimeFormat по умолчанию установлено в DateTimeFormat.ShowClock. Чтобы позволить пользователю переключаться между отображением даты и времени, измените класс MainWindow следующим образом (при этом также переключается метка пункта меню ToolStripDropDownButton):

```
public partial class MainWindow : Form
{
  // Какой формат отображать?
  DateTimeFormat dtFormat = DateTimeFormat.ShowClock;

  // Текущий помеченный элемент меню.
  private ToolStripMenuItem currentCheckedItem;
```

```
  public MainWindow()
  {
    InitializeComponent();
    // Эти свойства могут быть также установлены в окне Properties.
    Text = "Status Strip Example";
    CenterToScreen();
    BackColor = Color.CadetBlue;
    currentCheckedItem = currentTimeToolStripMenuItem;
    currentCheckedItem.Checked = true;
  }
...
  private void currentTimeToolStripMenuItem_Click(object sender, EventArgs e)
  {
    // Переключить метку и установить формат панели в отображение времени.
    currentCheckedItem.Checked = false;
    dtFormat = DateTimeFormat.ShowClock;
    currentCheckedItem = currentTimeToolStripMenuItem;
    currentCheckedItem.Checked = true;
  }
  private void dayoftheWeekToolStripMenuItem_Click(object sender, EventArgs e)
  {
    // Переключить метку и установить формат панели в отображение даты.
    currentCheckedItem.Checked = false;
    dtFormat = DateTimeFormat.ShowDay;
    currentCheckedItem = dayoftheWeekToolStripMenuItem;
    currentCheckedItem.Checked = true;
  }
}
```

## Отображение подсказок к пунктам меню

И, наконец, нужно сконфигурировать первую панель, содержащую строки подсказок меню. Как известно, большинство приложений посылают маленькие кусочки текстовой информации в первую панель линейки состояния всякий раз, когда пользователь выбирает пункт меню (вроде "This terminates the application" ("Завершить приложение")). Учитывая, что вы уже обрабатываете события MouseHover для каждого подменю в MenuStrip и ToolStripDropDownButton, все, что понадобится сделать — это присвоить правильное значение свойству Text переменной-члена toolStripStatusLabelMenuState. Например:

```
private void exitToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Exits the app."; }
private void aboutToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Shows about box."; }
private void dayoftheWeekToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Shows the day of the week."; }
private void currentTimeToolStripMenuItem_MouseHover(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Shows the current time."; }
```

Запустите обновленный проект на выполнение. Теперь вы должны увидеть эти строки подсказки в первой панели StatusStrip при выборе каждого из пунктов меню.

## Установка состояния "Ready"

Последнее, что осталось сделать с этим примером — обеспечить, что когда пользователь отменяет выделение пункта меню, первая текстовая панель отображает сообщение по умолчанию (например, "Ready" ("Готово")). При текущем дизайне подсказка к последнему выбранному пункту меню останется в левой текстовой панели, что в лучшем случае может запутать пользователя. Чтобы избавиться от этой неприятности, обработаем

событие MouseLeave для пунктов меню Exit, About, Day of Week (День недели) и Current Time (Текущее время). *Однако* вместо генерации нового обработчика события для каждого элемента позволим им всем вызывать метод по имени SetReadyPrompt():

```
private void SetReadyPrompt(object sender, EventArgs e)
{ toolStripStatusLabelMenuState.Text = "Ready."; }
```

После этого вы обнаружите, что первая панель возвращается к сообщению по умолчанию, как только курсор мыши покинет любой из четырех пунктов меню.

---

**Исходный код.** Проект StatusBarApp включен в подкаталог Bonus Chapter 2.

---

# Работа с элементом ToolStrip

Следующий элемент графического интерфейса уровня формы, который мы рассмотрим в этой главе — это тип .NET 2.0 ToolStrip, который призван заменить функциональность устаревшего класса .NET 1.x ToolBar. Как известно, панели инструментов обычно предоставляют альтернативные средства для активизации заданного элемента меню. Поэтому если пользователь щелкнет на кнопке Save (Сохранить), это должно иметь тот же эффект, как и выбор пункта меню File⇨Save. Подобно MenuStrip и StatusStrip, тип ToolStrip может содержать многочисленные элементы панели инструментов, и некоторые из них вы уже встречали в предыдущих примерах:

- ToolStripButton
- ToolStripLabel
- ToolStripSplitButton
- ToolStripDropDownButton
- ToolStripSeparator
- ToolStripComboBox
- ToolStripTextBox
- ToolStripProgressBar

Подобно другим элементам управления Windows Forms, ToolStrip поддерживает встроенный редактор, позволяющий быстро добавлять стандартные типы кнопок (File, Exit, Help, Copy (Копировать), Paste (Вставить) и т.п.) к ToolStrip, изменять позицию стыковки и встраивать ToolStrip в ToolStripContainer (подробнее — ниже). На рис. 2.20 показана поддержка элементов ToolStrip в дизайнере.

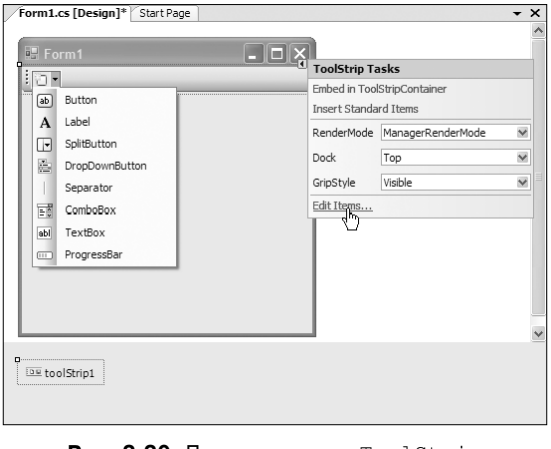


**Рис. 2.20.** Проектирование ToolStrip

Подобно MenuStrip и StatusStrip, индивидуальные элементы управления ToolStrip добавляются к его внутренней коллекции через свойство Items. Если щелкнуть на ссылке Insert Standard Items во встроенном редакторе ToolStrip, то метод InitializeComponent() будет обновлен кодом вставки массива типов-наследников ToolStripItem, представляющих каждый элемент:

```
private void InitializeComponent()
{
...
  // Автоматически сгенерированный код для подготовки ToolStrip.
  this.toolStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
    this.newToolStripButton, this.openToolStripButton,
    this.saveToolStripButton, this.printToolStripButton,
    this.toolStripSeparator, this.cutToolStripButton,
    this.copyToolStripButton, this.pasteToolStripButton,
    this.toolStripSeparator1, this.helpToolStripButton});
...
}
```

Чтобы проиллюстрировать работу с ToolStrips, следующее приложение Windows Forms создает ToolStrip, содержащий два элемента ToolStripButton (по имени toolStripButtonGrowFont и toolStripButtonShrinkFont), а также ToolBarSeparator и ToolBarTextBox (по имени toolStripTextBoxMessage).

Конечный пользователь может вводить сообщение, отображаемое в Form, через ToolBarTextBox, а два элемента ToolBarButton будут управлять увеличением и уменьшением размера шрифта. На рис. 2.21 показан конечный результат проекта.

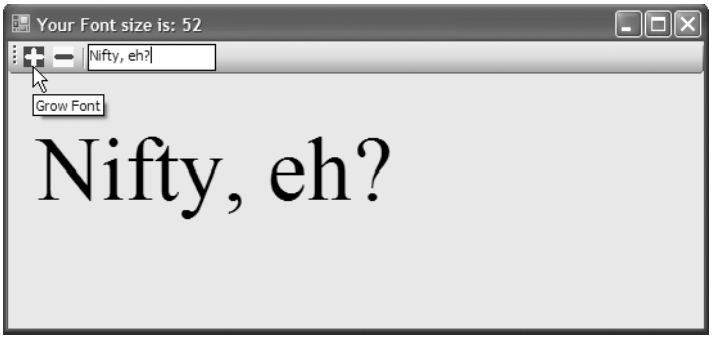


**Рис. 2.21.** ToolStrip в действии

По идее, вы уже достаточно хорошо освоили работу в дизайнере форм Visual Studio, так что погружаться в детали построения ToolStrip не имеет смысла. Обратите внимание, однако, что каждый экземпляр ToolStripButton имеет собственную пиктограмму, созданную в редакторе изображений Visual Studio. Если вы хотите создавать файлы изображений для проекта, выберите пункт меню Project⇨Add New Item (Проект⇨Добавить новый элемент) и в появившемся диалоговом окне добавьте новый файл пиктограммы (рис. 2.22).

После этого можно редактировать изображения, используя вкладку Colors (Цвета) панели инструментов Toolbox и панель инструментов Image Editor (Редактор изображений). В любом случае, как только вы создадите пиктограммы, их можно будет ассоциировать с элементами ToolStripButton через свойство Image в окне Properties. Добившись удовлетворительного внешнего вида ToolStrip, обработайте событие Click каждого ToolStripButton.

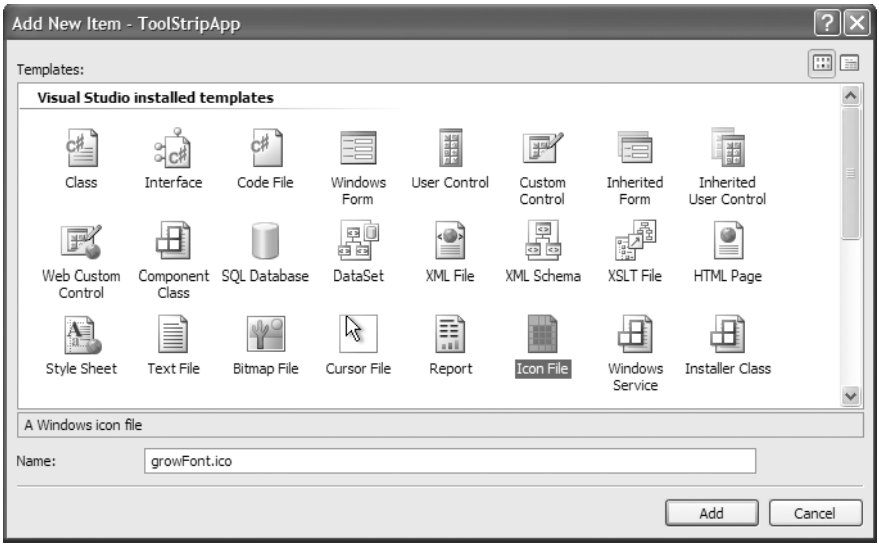


**Рис. 2.22.** Добавление новых файлов изображений

Ниже приведен соответствующий код метода `InitializeComponent()` для первого элемента `ToolStripButton` (второй `ToolStripButton` будет выглядеть почти так же):

```
private void InitializeComponent()
{
...
  // toolStripButtonGrowFont
  //
  this.toolStripButtonGrowFont.DisplayStyle =
     System.Windows.Forms.ToolStripItemDisplayStyle.Image;
  this.toolStripButtonGrowFont.Image =
     ((System.Drawing.Image)
     (resources.GetObject("toolStripButtonGrowFont.Image")));
  this.toolStripButtonGrowFont.ImageTransparentColor =
     System.Drawing.Color.Magenta;
  this.toolStripButtonGrowFont.Name = "toolStripButtonGrowFont";
  this.toolStripButtonGrowFont.Text = "toolStripButton2";
  this.toolStripButtonGrowFont.ToolTipText = "Grow Font";
  this.toolStripButtonGrowFont.Click += new
     System.EventHandler(this.toolStripButtonGrowFont_Click);
...
}
```

**На заметку!** Обратите внимание, что значение, присвоенное свойству `Image` объекта `ToolStripButton`, получено методом `GetObject()`. Этот метод применяется для извлечения встроенных ресурсов, используемых сборкой.

Остальной код исключительно прост. Обратите внимание, как в следующем обновленном коде `MainWindow` текущий размер шрифта ограничивается значениями между 12 и 70:

```
public partial class MainWindow : Form
{
  // Текущий, максимальный и минимальный размеры шрифта.
```

```
  int currFontSize = 12;
  const int MinFontSize = 12;
  const int MaxFontSize = 70;
  public MainWindow()
  {
    InitializeComponent();
    CenterToScreen();
    Text = string.Format("Your Font size is: {0}", currFontSize);
  }
  private void toolStripButtonShrinkFont_Click(object sender, EventArgs e)
  {
    // Уменьшить размер шрифта на 5 и обновить изображение.
    currFontSize -= 5;
    if (currFontSize <= MinFontSize)
      currFontSize = MinFontSize;
    Text = string.Format("Your Font size is: {0}", currFontSize);
    Invalidate();
  }
  private void toolStripButtonGrowFont_Click(object sender, EventArgs e)
  {
    // Увеличить размер шрифта на 5 и обновить изображение.
    currFontSize += 5;
    if (currFontSize >= MaxFontSize)
      currFontSize = MaxFontSize;
    Text = string.Format("Your Font size is: {0}", currFontSize);
    Invalidate();
  }
  private void MainWindow_Paint(object sender, PaintEventArgs e)
  {
    // Отобразить определенное пользователем сообщение.
    Graphics g = e.Graphics;
    g.DrawString(toolStripTextBoxMessage.Text,
        new Font("Times New Roman", currFontSize),
        Brushes.Black, 10, 60);
  }
}
```

В качестве последнего усовершенствования, если хотите, чтобы пользовательское сообщение обновлялось сразу после того, как ToolStripTextBox теряет фокус, можете обработать событие LostFocus и вызвать Invalidate() формы внутри сгенерированного обработчика события:

```
public partial class MainWindow : Form
{
...
  public MainWindow()
  {
  ...
    this.toolStripTextBoxMessage.LostFocus
        += new EventHandler(toolStripTextBoxMessage_LostFocus);
  }
  void toolStripTextBoxMessage_LostFocus(object sender, EventArgs e)
  {
    Invalidate();
  }
...
}
```

## Работа с элементом ToolStripContainer

ToolStrip при необходимости может быть сконфигурирован как элемент, стыкуемый с любой из сторон Form, содержащей его. Чтобы проиллюстрировать, как это делается, выполните щелчок правой кнопкой мыши на текущем ToolStrip в дизайнере и выберите в контекстном меню пункт Embed in ToolStripContainer (Встроить в ToolStripContainer). После этого вы обнаружите, что этот ToolStrip окажется внутри ToolStripContainer. Для данного примера выберите опцию Dock Fill in Form (Стыковать для заполнения формы), как показано на рис. 2.23.

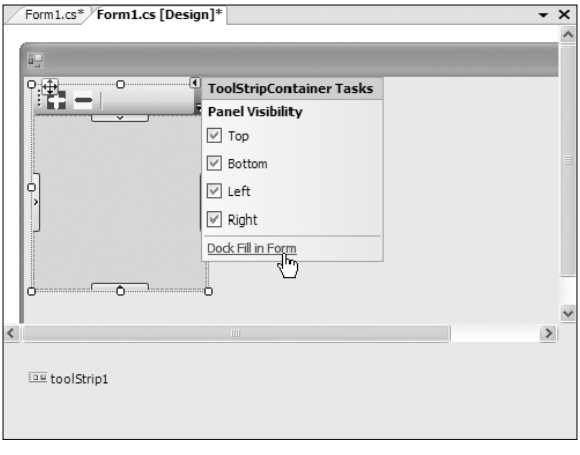


**Рис. 2.23.** Стыковка ToolStripContainer внутри всей формы

Если вы запустите программу с последними изменениями, то увидите, что ToolStrip можно перемещать и пристыковывать к любой стороне контейнера. Однако специальное сообщение теперь исчезло. Причина в том, что элементы ToolStripContainer являются дочерними элементами управления Form. Поэтому графическая визуализация происходит, но ее вывод скрыт контейнером, который теперь находится поверх клиентской области Form.

Для решения проблемы понадобится обработать событие Paint элемента ToolStripContainer вместо Form. Найдите событие Paint формы внутри окна Properties и выполните щелчок правой кнопкой мыши на текущем обработчике события. Выберите в контекстного меню пункт Reset (Сбросить), как показано на рис. 2.24.

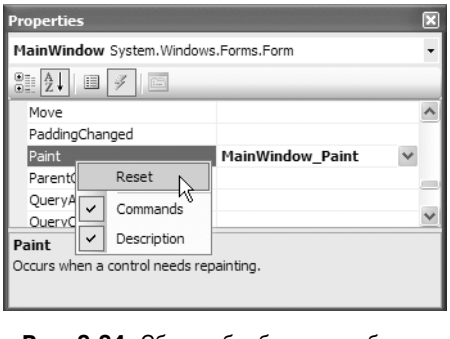


**Рис. 2.24.** Сброс обработчика события

Эту удалит логику обработчика события из InitializeComponent(), но оставит сам обработчик на месте (просто чтобы гарантировать, что вы не потеряете код, который хотели бы поддерживать).

Теперь обработайте событие Paint элемента ToolStripContainer и перенесите код визуализации из текущего обработчика события Paint формы в обработчик события Paint контейнера.

Сделав это, можно удалить (теперь пустой) метод MainWindow_Paint(). И, наконец, нужно заменить каждое вхождение вызова метода Invalidate() формы на вызов того же метода контейнера. Вот соответствующие изменения в коде:

```
public partial class MainWindow : Form
{
  ...
  void toolStripTextBoxMessage_LostFocus(object sender, EventArgs e)
  {
    toolStripContainer1.Invalidate(true);
  }

  private void toolStripButtonShrinkFont_Click(object sender, EventArgs e)
  {
  ...
    toolStripContainer1.Invalidate(true);
  }

  private void toolStripButtonGrowFont_Click(object sender, EventArgs e)
  {
  ...
    toolStripContainer1.Invalidate(true);
  }

  // Теперь мы перерисовываем контейнер, а не форму!
  private void ContentPanel_Paint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    g.DrawString(toolStripTextBoxMessage.Text,
        new Font("Times New Roman", currFontSize),
        Brushes.Black, 10, 60);
  }
}
```

Разумеется, ToolStripContainer может быть сконфигурирован различным образом, чтобы подкорректировать его поведение. Самостоятельно поищите в документации по .NET Framework 2.0 SDK необходимые детали. На рис. 2.25 показан готовый проект.

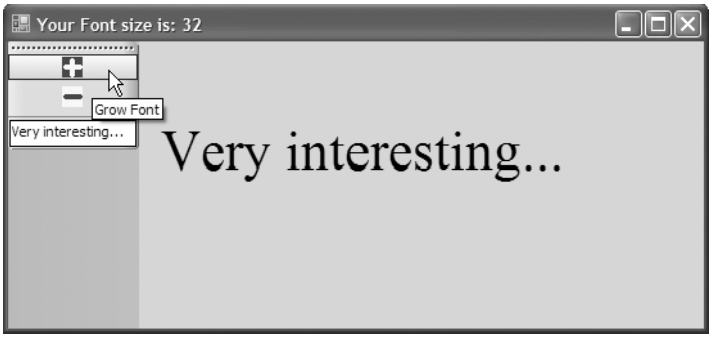


**Рис. 2.25.** Приложение ToolStripApp со стыкуемым ToolStrip

**Исходный код.** Проект ToolStripApp включен в подкаталог Bonus Chapter 2.

# Построение приложения MDI

В завершение начальной экскурсии по Windows Forms рассмотрим конфигурирование Form в качестве родителя для любого количества дочерних окон (т.е. в качестве контейнера MDI). Приложения MDI позволяют пользователям иметь одновременно несколько открытых дочерних окон в пределах окна верхнего уровня. В мире MDI каждое окно представляет определенный "документ" приложения. Например, Visual Studio — приложение MDI, потому что можно иметь открытыми несколько документов внутри одного экземпляра приложения.

При построении приложения MDI с помощью Windows Forms первая задача состоит (конечно) в создании нового приложения Windows. Начальная форма приложения обычно включает в себя систему меню, которое позволяет создавать новые документы (File⇨New (Файл⇨Создать)), а также организовать имеющиеся открытые дочерние окна (расположить каскадом, горизонтальной черепицей, вертикальной черепицей).

Создание дочернего окна интересно, поскольку вы обычно определяете прототип Form, который функционирует как основа для каждого нового дочернего окна. Учитывая, что формы являются типами классов, все приватные данные, определенные в дочерней форме, будут уникальны в пределах конкретного экземпляра. Например, если вы хотите создать приложение MDI — текстовый процессор, то можете создать дочернюю Form, поддерживающую StringBuilder для представления текста. Если пользователь создаст пять новых дочерних окон, каждое из них будет поддерживать свой собственный экземпляр StringBuilder, работающий независимо.

Вдобавок приложения MDI позволяют "объединять меню". Как упоминалось ранее, родительские окна обычно имеют систему меню, позволяющую пользователю порождать и организовывать дополнительные дочерние окна. Однако что если дочернее окно также имеет собственную систему меню? Если пользователь развернет во весь экран определенное дочернее окно, понадобится объединить систему меню этого дочернего окна с родительской формой, чтобы позволить пользователю работать с элементами обеих систем меню. В пространстве имен Windows Forms определено множество свойств, методов и событий, которые позволяют программно объединять системы меню. Вдобавок, существует система "стандартного объединения", которая подходит во многих случаях.

## Построение родительской формы

Чтобы ознакомиться с основами построения приложений MDI, начните с создания нового приложения Windows по имени SimpleMdiApp. Почти вся инфраструктура MDI может быть присвоено начальной форме посредством различных инструментов дизайна. Для начала найдите свойство IsMdiContainer в окне Properties и установите его в true. Если вы посмотрите на форму в дизайнере, то увидите, что после этого клиентская область изменится, визуально представляя контейнер для дочерних окон.

Затем поместите на форму новый элемент MenuStrip. Это меню специфицирует три элемента верхнего уровня — File (Файл), Window (Окно) и Arrange Windows (Упорядочить окна). Меню File содержит два подменю по имени New (Создать) и Exit (Выход). Меню Window не содержит никаких подэлементов, поскольку они будут добавляться программно по мере создания пользователем дополнительных дочерних окон. И, наконец, меню Arrange Windows определяет три подэлемента по имени Cascade (Каскадом), Vertical (Вертикально) и Horizontal (Горизонтально).

Создав пользовательский интерфейс меню, обработайте события Click для пунктов меню Exit, New, Cascade, Vertical и Horizontal (меню Window пока подэлементов не имеет). Обработчик File⇨New будет реализован в следующем разделе, а пока ниже показан код, обрабатывающий остальные пункты меню:

```
// Обработка события File | Exit и организация дочерних окон.
private void cascadeToolStripMenuItem_Click(object sender, EventArgs e)
{
   LayoutMdi(MdiLayout.Cascade);
}
private void verticalToolStripMenuItem_Click(object sender, EventArgs e)
{
   LayoutMdi(MdiLayout.TileVertical);
}
private void horizontalToolStripMenuItem_Click(object sender, EventArgs e)
{
   LayoutMdi(MdiLayout.TileHorizontal);
}
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
   Application.Exit();
}
```

Основной момент, представляющий для нас интерес — использование метода LayoutMdi() и соответствующее перечисление MdiLayout. Код обработки выбора каждого пункта меню должен быть достаточно ясен. Когда пользователь выбирает определенную компоновку, вы просто указываете родительской форме, чтобы она автоматически позиционировала все дочерние окна надлежащим образом.

Прежде чем перейти к конструированию дочерней формы, необходимо установить одно дополнительное свойство MenuStrip. Свойство MdiWindowListItem применяется для установки пункта меню верхнего уровня, который должен использоваться для автоматического перечисления списка имен каждого дочернего окна в виде возможных выборов меню. Установите это свойство в переменную-член windowToolStripMenuItem. По умолчанию этот список будет показывать значение свойства Text дочернего окна, за которым следует числовой суффикс (т.е. Form1, Form2, Form3 и т.д.).

## Построение дочерней формы

Теперь, имея контейнер MDI, нужно создать дополнительную форму, которая послужит прототипом для дочерних окон. Начните с добавления нового типа Form к текущему проекту (используя пункт меню Project⇨Add Windows Form (Проект⇨Добавить форму Windows)) по имени ChildPrototypeForm и обработайте событие Click для этой формы. В сгенерированном обработчике события случайным образом установите цвет фона клиентской области. Вдобавок выведите строковое представление нового Color в заголовок дочернего окна. Следующая логика должна выполнить этот трюк:

```
private void ChildPrototypeForm_Click(object sender, EventArgs e)
{
  // Получить три случайных числа.
  int r, g, b;
  Random ran = new Random();
  r = ran.Next(0, 255);
  g = ran.Next(0, 255);
  b = ran.Next(0, 255);
  // Создать цвет фона.
  Color currColor = Color.FromArgb(r, g, b);
  this.BackColor = currColor;
  this.Text = currColor.ToString();
}
```

## Порождение дочерних окон

И последняя задача в рассматриваемом примере — реализация обработчика пункта меню родительской формы File⇨New. При наличии определения дочерней формы логика проста: создать и показать новый экземпляр типа `ChildPrototypeForm`. К тому же, потребуется установить значение свойства `MdiParent` дочерней формы, чтобы оно указывало на включающую родительскую форму — главное окно. Вот необходимые дополнения:

```
private void newToolStripMenuItem_Click(object sender, EventArgs e)
{
  // Создать новое дочернее окно.
  ChildPrototypeForm newChild = new ChildPrototypeForm();

  // Установить родительскую форму для дочернего окна.
  newChild.MdiParent = this;

  // Отобразить новую форму.
  newChild.Show();
}
```

**На заметку!** Дочерняя форма может обращаться к свойству `MdiParent` напрямую всякий раз, когда ей надо манипулировать (или взаимодействовать) родительским окном.

Чтобы протестировать это приложение, начните с создания набора дочерних окон и пощелкайте в каждом из них, чтобы установить уникальный цвет фона. Если вы взглянете на пункты меню Window, то обнаружите там представление каждой дочерней формы. К тому же, если вы обратитесь к пунктам меню Arrange Windows, то сможете проинструктировать родительскую форму, чтобы она различным образом расположила открытые дочерние формы. На рис. 2.26 показано готовое приложение.

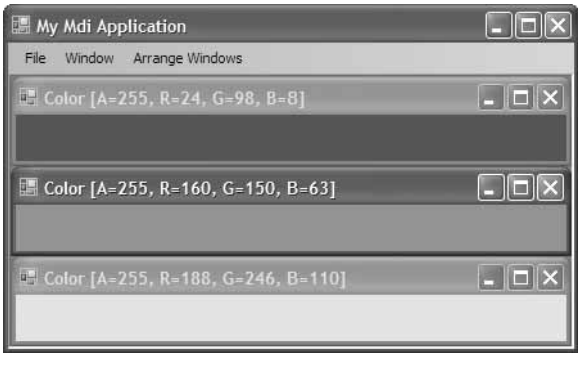


**Рис. 2.26.** Приложение MDI

**Исходный код.** Проект `SimpleMdiApp` находится в подкаталоге `Bonus Chapter 2`.

# Резюме

В этой главе было продемонстрировано построение пользовательского интерфейса с помощью типов из пространства имен System.Windows.Forms. Вы начали с построения нескольких приложений вручную и узнали, что приложение GUI, как минимум, нуждается в классе-наследнике Form и методе Main(), вызывающем Application.Run().

На протяжении этой главы вы научились строить меню верхнего уровня (и всплывающие меню), а также узнали, как реагировать на различные события меню. Вы узнали, как усовершенствовать типы Form с помощью панелей инструментов и линеек состояния. Как вы видели, в .NET 2.0 предпочтительно строить такие элементы пользовательского интерфейса посредством классов MenuStrips, ToolStrips и StatusStrips вместо их предшественников из .NET 1.x — MainMenu, ToolBar и StatusBar (хотя эти устаревшие типы по-прежнему поддерживаются). И, наконец, глава завершилась иллюстрацией конструирования приложения MDI с использованием Windows Forms.