Глава 1. Разработка программ в среде Microsoft Visual Studio .NET

Описания приемов работы ориентированы на два варианта интегрированных сред, разработанных корпорацией Microsoft. Первым вариантом является свободно распространяемая бесплатная система Microsoft Visual C# Express Edition, вторым вариантом является коммерческая система Microsoft Visual Studio, позволяющая разрабатывать приложения на различных языках платформы .NET. В среде Microsoft Visual Studio предусмотрены средства для разработки существенно большего числа типов приложений, однако в отношении стандартных Windows-приложений возможности данной среды по существу совпадают с возможностями ее экспресс-варианта для языка C#.

В настоящее время наиболее распространенными версиями как системы Microsoft Visual C# Express Edition, так и системы Microsoft Visual Studio является версия 2005, ориентированная на платформу .NET 2.0, и версия 2008, ориентированная на платформу .NET 3.5. Во всех проектах рассматриваются возможности, имеющиеся в обеих версиях.

В дальнейшем под средой Visual C# мы будем подразумевать как среду Microsoft Visual C# 2005/2008, входящую в состав системы Microsoft Visual Studio 2005/2008, так и ее экспресс-вариант.

1.1. Создание, сохранение и открытие проекта

При запуске среды Visual C# в нее автоматически загружается стартовая страница (**Start Page**), позволяющая быстро загрузить один из ранее разрабатывавшихся проектов (список **Recent Projects**), открыть какой-либо другой существующий проект (пункт **Open: Project…**), а также создать новый проект (пункт **Create: Project…**). Впрочем, все отмеченные возможности доступны также из меню среды Visual C#:

- **File | New | Project…** или <Ctrl>+<Shift>+<N> — создание нового проекта (в экспресс-варианте данная команда является командой меню второго уровня: **File | New Project…**);
- **File | Open | Project/Solution…** или комбинация <Ctrl>+<Shift>+<O> — открытие существующего проекта (в экспресс-варианте команда имеет вид **File | Open Project…**);
- **File | Recent Projects** — загрузка одного из недавно разрабатывавшихся проектов.

Среда Visual C# организована таким образом, что в ней нельзя создать "отдельно взятый" проект. Каждый проект должен содержаться в особой сущности, называемой solution ("решение"), которую можно охарактеризовать как *группу взаимосвязанных проектов* (по этой причине термин solution мы будем переводить как "группа проектов"). В каждый момент времени в среду Visual C# можно загрузить только одну группу проектов; загрузка другой группы приводит к автоматическому закрытию предыдущей.

При создании нового проекта на экране возникает диалоговое окно, в котором надо выбрать тип проекта и его имя. В качестве типа для всех рассматриваемых проектов (кроме двух первых проектов DISKINFO и EXCEP), следует выбирать **Windows Application**; в качестве имени рекомендуется указывать имя примера, приведенное в заголовке соответствующей главы, например, EVENTS (гл. 4). Заметим, что имя проекта может содержать не только цифры и латинские буквы, но и другие символы, допустимые в именах файлов, в том числе пробелы и русские буквы, хотя этой возможностью не следует злоупотреблять.

В среде Visual Studio при создании нового проекта необходимо сразу указать каталог для его сохранения (поле **Location**). На размещение проекта также влияет информация, связанная с группой проектов (solution), в которую будет помещен создаваемый проект, в частности, флажок **Create directory for solution** (Создать каталог для группы проектов). Если в группу будет входить единственный проект, то следует снять данный флажок; это приведет к тому, что созданная группа проектов будет иметь то же имя, что и созданный проект, а в каталоге, указанном в поле **Location**, будет создан каталог с именем проекта, в котором будут размещаться все файлы, связанные с проектом и его группой.

При установленном флажке **Create directory for solution** можно указать имя группы проектов, отличное от имени самого проекта. В этой ситуации создается более сложная иерархия каталогов: в каталоге, указанном в поле **Location**, будет создан каталог с именем группы проектов, а в нем — каталог с именем проекта.

Отметим еще одну возможность среды Visual Studio (которой мы, однако, пользоваться не будем): создаваемый проект можно *добавить* к текущей группе проектов, то есть группе, ранее загруженной в среду Visual Studio.

В экспресс-варианте Visual C# при создании нового проекта не требуется указывать его расположение; вся необходимая информация запрашивается при первом *сохранении* созданного проекта, поэтому рекомендуется выполнить это сохранение сразу после создания проекта.

Для сохранения всех изменений, внесенных в текущий проект (а точнее, в текущую группу проектов), в Visual C# предусмотрена команда **File | Save All**, а также клавиатурная комбинация <Ctrl>+<Shift>+<S>.

При открытии существующего проекта на экране появляется диалоговое окно **Open**, в котором можно выбрать либо файл с расширением .sln (содержащий информацию о группе проектов с указанным именем), либо файл с расширением .csproj (содержащий информацию о проекте с указанным именем). Однако даже при выборе отдельного проекта загружается группа проектов, в которой он содержится.

Некоторые примеры предполагают не создание нового проекта "с нуля", а модификацию уже имеющегося проекта. Имена таких примеров снабжены порядковыми номерами, начиная с номера 2 (например, PNGEDIT2–PNGEDIT4). Перед началом выполнения подобных примеров следует скопировать в новый каталог все файлы проекта из примера с предыдущим номером, после чего загрузить полученную копию проекта в среду Visual C# и начать ее модификацию.

Действия по модификации проекта опишем на примере преобразования проекта PNGEDIT1 в проект PNGEDIT2.

Вначале следует изменить имя проекта и имя связанной с ним группы проектов. Это проще всего выполнить с помощью окна **Solution Explorer**, обычно расположенного в правой части экрана (на рис. 1.1 слева приведен вид этого окна для проекта PNGEDIT1): достаточно выделить в окне **Solution Explorer** строку, соответствующую проекту (на рис. 1.1 эта строка является выделенной) или группе проектов (данная строка начинается со слова Solution), нажать клавишу <F2>, ввести новое имя и нажать <Enter>. Вид окна **Solution Explorer** после изменения имени проекта PNGEDIT1 на PNGEDIT2 приводится на рис. 1.1 справа.

Следует также изменить имя сборки (assembly name), то есть имя результирующего exe-файла. Для этого надо выполнить команду **Project |** *<Имя проекта>* **Properties** (в нашем случае **Project | PNGEDIT2 Properties**), перейти в появившейся вкладке с именем проекта в раздел настроек **Application** и указать новое имя в поле **Assembly name** (см. рис. 1.2; для версии 2008 вид раздела **Application** несколько отличается от приведенного на рисунке). Можно также изменить имя пространства имен по умолчанию (поле **Default namespace**), однако в этом нет необходимости.

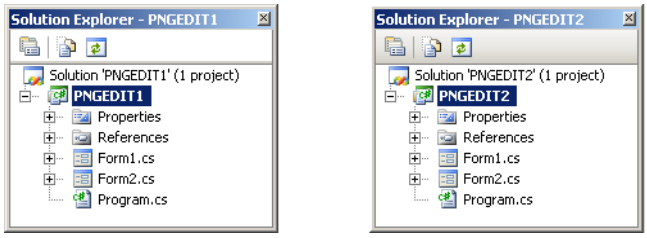


Рис. 1.1. Вид окна Solution Explorer до изменения имени проекта (слева) и после его изменения (справа)

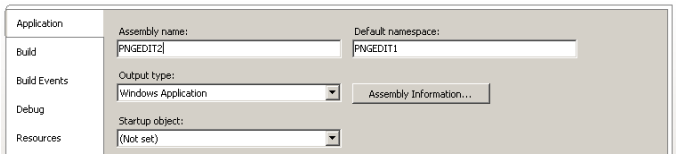


Рис. 1.2. Вид верхней части окна со свойствами проекта для версии Visual Studio 2005

1.2. Добавление к проекту новой формы и размещение в форме нового компонента

При создании нового проекта типа Windows Application связанная с ним главная форма (класс с именем `Form1`) создается автоматически и сразу загружается в среду Visual C#. С формой `Form1` связывается набор файлов, основными из которых являются файлы Form1.cs и Form1.Designer.cs. В этих файлах содержится описание данной формы на языке C#, причем второй из них содержит ту часть описания, которая генерируется автоматически в ответ на различные действия разработчика, связанные с визуальным проектированием (собственный код разработчик обычно размещает в файле Form1.cs).

Если в проекте требуется использовать дополнительные формы (см., например, гл. 5), то их проще всего добавить в проект с помощью команды меню **Project | Add Windows Form…** . При выполнении этой команды возникает диалоговое окно, в котором надо указать имя добавляемой формы (которое одновременно будет и начальной частью имен файлов, содержащих ее описание). В описываемых примерах всегда используются имена форм, предлагаемые системой по умолчанию (`Form1`, `Form2` и т. д.).

Любая созданная форма отображается в редакторе Visual C# в *режиме дизайна* (на экране появляется изображение формы и ее содержимого); соответствующая вкладка редактора заканчивается текстом **[Design]**. Например, для главной формы `Form1` вкладка содержит текст **Form1.cs [Design]**. Для перехода к содержимому соответствующего cs-файла достаточно нажать клавишу <F7>; при этом в редакторе появится новая вкладка с текстом данного файла (ярлычок этой вкладки будет содержать имя файла, например, **Form1.cs**). Для обратного переключения с текста на изображение формы можно использовать комбинацию <Shift>+<F7>. Переключаться между изображением формы и текстом кода можно также с помощью *контекстного меню*: при щелчке правой кнопкой мыши на изображении формы первый пункт появляющегося меню имеет имя **View Code** и позволяет перейти к тексту cs-файла, а при вызове контекстного меню для текста cs-файла первый пункт меню имеет имя **View Designer** и позволяет вернуться в режим дизайна.

Другим удобным способом переключения между различными окнами среды Visual C# является использование комбинации <Ctrl>+<Tab>: после ее нажатия на экране появляется вспомогательное окно со списком всех загруженных файлов и форм (Active Files), а также всех открытых вспомогательных окон (Active Tool Windows). На рис. 1.3 приведен примерный вид вспомогательного окна для версии 2005 (в версии 2008 окно имеет незначительные отличия). После появления вспомогательного окна надо, не отпуская клавиши <Ctrl>, выделить имя файла или окна, которое надо активизировать (для выбора можно использовать клавиши со стрелками и клавишу <Tab>), а затем отпустить клавишу <Ctrl>.

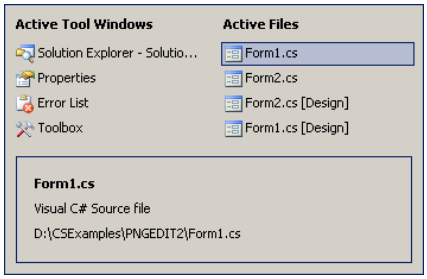
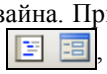


Рис. 1.3. Вспомогательное окно для выбора одного из загруженных файлов или окон среды Visual C#

Любую вкладку в редакторе можно закрыть; для этого достаточно сделать ее активной и нажать комбинацию <Ctrl>+<F4> или вызвать контекстное меню вкладки, щелкнув правой кнопкой мыши на ее ярлычке, и выбрать пункт **Close.**

Если после загрузки существующего проекта в редактор не загрузилась требуемая форма, достаточно выполнить двойной щелчок мышью на имени этой формы в окне **Solution Explorer** (см. рис. 1.1). Можно также щелкнуть на имени этой формы *правой* кнопкой мыши и выбрать пункт **Open** из появившегося контекстного меню. Данное меню также содержит пункт **View Code**, позволяющий загрузить в редактор текст cs-файла, и **View Designer**, позволяющий загрузить в редактор форму в режиме дизайна. При выборе одной из форм в окне проекта **Solution Explorer** дополнительно появляются кнопки быстрого доступа  , первая из которых позволяет загрузить текст файла, а вторая — отобразить форму в режиме дизайна.

Для размещения в форме нового компонента следует воспользоваться *окном компонентов* **Toolbox** (отобразить это окно на экране можно либо командой меню **View | Toolbox**, либо комбинацией клавиш <Ctrl>+<Alt>+<X>, либо кнопкой ). Это окно обычно отображается в левой части экрана; на рис. 1.4 приведен вид верхней части окна **Toolbox** при выборе в нем группы **All Windows Forms**. Заметим, что компоненты отображаются в окне **Toolbox** только в том случае, если редактор находится в режиме дизайна. Надо выбрать в этом окне группу, содержащую нужный компонент (щелкнув мышью на имени этой группы), затем щелкнуть на изображении нужного компонента, выделив его (на рис. 1.4 выделен компонент `Button`), и, наконец, щелкнуть в том месте формы, где предполагается разместить выбранный компонент.

Для того чтобы выбрать компонент в окне **Toolbox**, необязательно знать, в какой группе он содержится, поскольку в этом окне имеется группа **All Windows Forms**, в которой указаны все имеющиеся компоненты в алфавитном порядке (см. рис. 1.4).

Если в окне **Toolbox** отсутствует нужный компонент, то можно попытаться загрузить его в это окно. Действия, которые при этом надо выполнить, описываются в разд. 13.1.

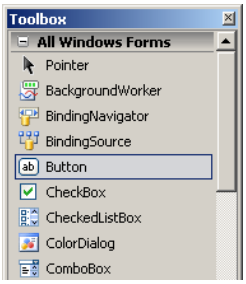


Рис. 1.4. Вид верхней части окна Toolbox

Для быстрого размещения в форме нескольких компонентов одного типа следует после выделения требуемого компонента в окне **Toolbox** выполнить щелчок мышью на форме несколько раз, держа нажатой клавишу <Ctrl>. Если в окне **Toolbox** требуется отменить выбор компонента, не размещая его в форме, то достаточно выделить в этом окне элемент **Pointer** (он располагается первым в любой группе компонентов; рядом с ним изображен курсор в виде стрелки — см. рис. 1.4).

Если компонент требуется разместить не в форме, а в другом компоненте (например, в компоненте `Panel`), то при размещении компонента необходимо выполнить щелчок мыши в области компонента-приемника. В качестве компонентов-приемников могут использоваться только особые компоненты, называемые *компонентами-контейнерами* (сама форма также является компонентом-контейнером). Компонент-контейнер (форма, панель и т. д.), содержащий другие компоненты, называется *родительским* (parent) по отношению к размещенным в нем *дочерним* компонентам.

При описании этапов разработки проекта мы всегда будем указывать, в каком компоненте-контейнере следует разместить каждый компонент.

После добавления компонента в форму он автоматически получает *имя*, которое сохраняется в его свойстве `Name`. По умолчанию имя любого компонента начинается со строчной буквы и состоит из имени типа компонента и порядкового номера. Так, первая размещенная в форме кнопка (компонент типа `Button`) получит имя `button1`, а вторая кнопка — имя `button2`. Формы имеют имена, начинающиеся с прописной буквы (`Form1`, `Form2` и т. д.); это связано с тем, что данные имена являются именами новых *классов* — потомков базового класса `Form`. Имя, присвоенное по умолчанию, всегда можно заменить на другое имя, позволяющее, например, уточнить назначение того или иного компонента (например, кнопку `button1`, содержащую текст `OK`, можно назвать `btnOK`). Однако в описываемых проектах почти всегда используются имена компонентов, предлагаемые системой Visual C# по умолчанию: это позволяет уменьшить количество действий по настройке свойств компонентов и упрощает ориентировку в текстах программ. "Значимые" имена используются только для пунктов меню, кнопок быстрого доступа и разделов статусной панели (см. гл. 18, 21, 22). Следует заметить, однако, что при разработке больших проектов целесообразно использовать значимые имена для всех компонентов.

При описании действий по добавлению компонента в форму почти никогда не уточняется, как именно *позиционировать* компонент на его родительском компоненте, поскольку это легко определить по приводимому рисунку. Перечислим способы позиционирования компонента:

- перетаскивание мышью по форме (при этом на форме появляются вспомогательные *линии выравнивания*, позволяющие осуществить привязку компонента к границам формы или разместить его на уровне границ существующих компонентов);
- использование панели выравнивания **Layout** (см. разд. 9.1);

- перемещение на один пиксел с помощью клавиш со стрелками;
- явное указание значения свойства `Location` в окне **Properties** (работа с окном **Properties** подробно рассматривается в разд. 1.3).

Размеры визуальных компонентов также обычно не уточняются. Для настройки размеров можно воспользоваться перетаскиванием мышью за один из маркеров, окружающих выделенный компонент, или клавишами со стрелками при нажатой клавише <Shift>. Можно также явно задать значения свойства `Size` в окне **Properties** или воспользоваться панелью **Layout**.

Дополнительные сведения, связанные с настройкой меню приложения, его панели инструментов и статусной панели, приводятся в гл. 18, 21 и 22.

1.3. Настройка свойств форм и компонентов

Для настройки свойств форм и компонентов используется *окно свойств* **Properties**, быстро перейти на которое можно с помощью комбинации <Alt>+<Enter> (вернуться обратно на форму можно с помощью комбинации <Ctrl>+<Tab>). Окно **Properties** обычно располагается в правом нижнем углу экрана и может находиться в двух режимах: **Properties** и **Events** (см. рис. 1.5) . В настоящем разделе мы рассмотрим режим **Properties**, предназначенный для отображения *свойств* выделенного компонента или группы компонентов (для перехода в данный режим надо нажать третью кнопку  на панели инструментов окна свойств).

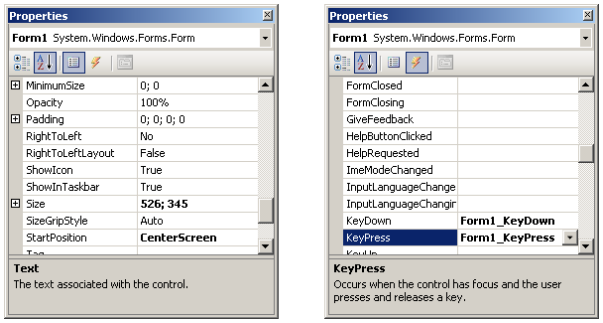


Рис. 1.5. Окно Properties в режимах Properties (слева) и Events (справа)

Если выделена группа компонентов, то в окне свойств отображаются только те свойства, которые имеются у всех выделенных компонентов.

Выделение группы компонентов позволяет быстро задать для них одинаковые размеры, заголовки или другие общие свойства, а также переместить их, сохраняя взаимное расположение ("как одно целое"). Перечислим способы выделения группы компонентов:

- щелчок на компонентах при нажатой клавише <Shift> или <Ctrl>;
- охват компонентов пунктирной рамкой, которая появляется на форме при перемещении мыши с нажатой левой кнопкой (для выделения достаточно захватить рамкой часть компонента).

Если выделена группа компонентов, то один компонент этой группы является *текущим* (его маркеры имеют белый цвет). На рис. 1.6 приведен фрагмент формы с четырьмя выделенными компонентами; текущим является компонент `button2`. Результат некоторых действий (например, связанных с выравниванием компонентов на форме — см. разд. 9.1) зависит от того, какой компонент выделенной группы является текущим. Для того чтобы сделать текущим другой компонент из выделенной группы, достаточно выполнить на нем щелчок мышью. Для снятия выделения с группы компонентов надо выделить компонент, не входящий в эту группу.

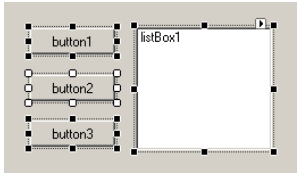


Рис. 1.6. Фрагмент формы с группой выделенных компонентов

Полезно знать, что если выделен некоторый компонент, то для выделения его родительского компонента достаточно нажать клавишу <Esc> (таким способом можно быстро выделить любой компонент-контейнер, даже если его целиком покрывают дочерние компоненты). Для выделения формы есть еще более простой способ: щелчок мышью на ее *заголовке*.

С помощью двух первых кнопок окна свойств  (см. рис. 1.5) можно настроить способ отображения свойств: по категориям (первая кнопка) или в алфавитном порядке имен (вторая кнопка). Каждая категория имеет имя (например, **Appearance**) и содержит "родственные" свойства. Используя группировку свойств по алфавиту, проще перейти к требуемому свойству; в то же время, при начальном ознакомлении со свойствами, имеющимися у компонента, удобнее использовать группировку по категориям.

В примерах настройка свойств форм и компонентов описывается в листингах, озаглавленных "Настройка свойств". Примером подобного листинга является листинг 1.1 — фрагмент листинга 5.1.

В листингах настройки свойств вначале указывается имя компонента, свойства которого требуется изменить, а затем, после двоеточия, — список его настраиваемых свойств в формате

`имя свойства = **новое значение свойства**`

(значение свойства выделяется полужирным шрифтом). Свойства перечисляются через запятую.

Компоненты могут иметь *составные свойства*, то есть свойства, имеющие собственные свойства (например, `Font`). Если требуется настроить одно или несколько свойств подобного составного свойства, в листинге настройки свойств используется разделитель точка (см. листинг 1.2, являющийся копией листинга 30.3). Составные свойства помечаются в окне **Properties** знаком "+" слева от имени (на рис. 1.5 знаком "+" помечены свойства `MinimumSize`, `Padding` и `Size`). Щелчок на знаке "+" приводит к отображению в окне всех свойств выбранного составного свойства; знак "+" при этом меняется на "–". Щелчок на знаке "–" сворачивает список свойств составного свойства.

Листинг 1.2 демонстрирует еще одно используемое обозначение: если требуется очистить какое-либо свойство, то в качестве его значения указывается курсивный текст "*пустая строка*".

Для значений логических свойств в листингах "Настройка свойств" используются константы **`True`** и **`False`**, начинающиеся с заглавной буквы (в отличие от ключевых слов языка C# `true` и `false`); это связано с тем, что именно так указываются логические константы в окне **Properties.**

Листинг 1.1. Настройка свойств

```
Form1: Text = Главное окно, MaximizeBox = False,
  FormBorderStyle = FixedSingle
button1: Text = Открыть подчиненное окно
button2: Text = Открыть диалоговое окно
```

Листинг 1.2. Настройка свойств

```
Form2: Text = пустая строка, ControlBox = False,
  FormBorderStyle = FixedSingle, Opacity = 80%,
  ShowInTaskbar = False,
  StartPosition = CenterScreen,
  UseWaitCursor = True
label1: Text = Тригонометрические функции,
  AutoSize = False, Dock = Fill,
  TextAlign = MiddleCenter,
  Font.Name = Times New Roman, Font.Size = 32,
  Font.Bold = True
```

Для задания некоторых свойств, связанных с привязкой или выравниванием (например, `Dock` или `TextAlign`), в окне **Properties** используются *вспомогательные панели*. Для отображения таких панелей предназначена кнопка , появляющаяся при выделении соответствующего свойства. В этих панелях надо выбрать один или несколько элементов, расположенных в требуемой позиции (слева, справа, по центру и т. п.), причем результат выбора будет отображен в окне **Properties** в виде обычного текста (например, **`Bottom`** или **`MiddleCenter`**). Хотя подобные действия являются интуитивно понятными, они, как правило, снабжаются в тексте примеров дополнительными пояснениями.

В некоторых случаях для задания свойства бывает удобно воспользоваться специальным *диалоговым окном*. Если для свойства предусмотрено такое окно, то при выделении свойства в окне **Properties** справа от него изображается кнопка с многоточием  , позволяющая вызвать диалоговое окно (примером такого свойства является `Font`).

1.4. Определение обработчиков событий

У всех компонентов имеется не только набор свойств, но и набор *событий*, с которыми можно связать методы — *обработчики событий*. Список обработчиков для выделенного компонента или группы компонентов отображается в окне свойств в режиме **Events** (см. рис. 1.5); для перехода в данный режим надо нажать четвертую кнопку  на панели инструментов окна **Properties**. Список событий, как и список свойств, можно упорядочивать двумя способами: по категориям и по именам (в алфавитном порядке).

Двойной щелчок мышью на *пустом* поле ввода рядом с именем нужного события обеспечивает автоматическое создание заготовки для обработчика этого события. Обработчик любого события для любого компонента оформляется как метод той формы, на которой расположен данный компонент. Во всех примерах используются имена методов-обработчиков, предлагаемые системой Visual C# по умолчанию: это позволяет легко определить, с каким компонентом и каким событием связан обработчик.

Тексты обработчиков приводятся в в листингах, заголовок которых начинается со слова "Обработчик" (см. листинг 1.3, являющийся копией листинга 4.3).

Листинг 1.3. Обработчик `Form1.MouseDown`

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
  button1.Location = new Point(e.X - button1.Width / 2,
    e.Y - button1.Height / 2);
}
```

Полужирным шрифтом выделяются те строки, которые требуется добавить в автоматически созданную заготовку метода-обработчика. Иногда фрагменты программы снабжаются комментариями, хотя чаще пояснения даются в основном тексте

примера или в специальном разделе "Комментарий". Символ ↳ обозначает продолжение предыдущей строки программы, которое *не следует* набирать на новой строке.

Если в качестве обработчика нужно указать имя *уже имеющегося* метода, то достаточно выбрать имя этого метода в выпадающем списке рядом с именем события в окне свойств (кнопка разворачивания списка появляется только около выделенного события; на рис. 1.5 справа эта кнопка изображена рядом с событием `KeyPress`).

Так как при разработке программ в Visual C# необходимо иметь четкое представление о том, как создаются и используются обработчики событий, этой теме посвящен специальный проект EVENTS (гл. 4). Связывание обработчика с несколькими событиями подробно обсуждается в гл. 6.

1.5. Внесение изменений в текст программы

При выполнении примеров изменения, как правило, вносятся в текст уже имеющихся обработчиков. Если изменения являются существенными и охватывают весь текст обработчика, то приводится его новый полный текст, в котором измененные или добавленные строки или фрагменты строк выделяются полужирным шрифтом (см. листинг 1.4, являющийся копией листинга 6.7).

Листинг 1.4. Новый вариант метода `button1_Click`

```
private void button1_Click(object sender, EventArgs e)
{
  label1.Text = (sender as Button).Text[1].ToString();
  label2.Text = "=";
}
```

Если же изменения незначительны, а обработчик достаточно велик, то просто указывается, какие операторы надо добавить или заменить, например:

в метод `SaveToFile` формы `Form1` добавьте оператор

```
textBox1.Modified = false;
```

В методе `button2_Click` формы `Form1` вставьте после оператора

```
Image im = new Bitmap(s);
```

два следующих оператора:

```
Graphics g = Graphics.FromImage(im);
g.Dispose();
```

Если место добавления не уточняется, то операторы должны добавляться *в конец* указанного метода.

Перейти к уже имеющемуся обработчику для его корректировки можно, либо перемещаясь по тексту файла, либо с помощью окна свойств, выполнив двойной щелчок мышью на поле ввода с именем нужного обработчика.

Аналогичным образом описываются изменения текста программы, не связанные с конкретным обработчиком, например:

в описание класса `Form1` добавьте поле

```
private Form2 form2 = new Form2();
```

В конструктор класса `Form1` добавьте оператор:

```
AddOwnedForm(form2);
```

Примечание

В качестве русского эквивалента английского термина operator используется термин "операция", например, "операция +", в то время как термин "оператор" применяется для обозначения "программных инструкций" (statements).

При наборе и редактировании текста программ следует использовать дополнительные возможности редактора Visual C#. Опишем некоторые из них.

1. Если ввести имя объекта (например, `button1`) и точку после него, то на экране появится список всех методов и свойств, которые имеет данный объект, причем для быстрого перехода к нужному методу достаточно набрать несколько его первых символов. Для вставки в текст программы имени выбранного метода или свойства надо нажать клавиши <Enter>, <Tab> или <пробел>. Список методов и свойств можно вызвать и явным образом, нажав комбинацию <Ctrl>+<пробел>.
2. После ввода имени метода и скобки (на экране появляется подсказка с кратким описанием этого метода и списком всех его параметров. Если метод является *перегруженным*, то есть может вызываться с различным набором параметров, то можно просмотреть все его перегруженные варианты, нажимая клавиши <↑> и <↓>. Для вызова подобной подсказки можно также нажать комбинацию <Ctrl>+<Shift>+<пробел>.
3. Для быстрого перехода к нужному фрагменту программы удобно использовать *закладки* (bookmarks). Для установки/отмены закладки на текущей строке программы (то есть строке, содержащей клавиатурный курсор) надо нажать комбинацию <Ctrl>+<B> и затем клавишу <T>, а для перехода к следующей или предыдущей установленной закладке надо нажать <Ctrl>+<B> и затем <N> или <P> соответственно. Можно также использовать меню **Edit | Bookmarks** и кнопки быстрого доступа  на панели **Text Editor**.
4. Если требуется *закомментировать* какой-либо фрагмент программы, то достаточно выделить его и нажать комбинацию <Ctrl>+<E>, а затем клавишу <C>. Для того чтобы раскомментировать закомментированный фрагмент, надо выделить его и нажать <Ctrl>+<E>, а затем <U>. Если требуется закомментировать или раскомментировать одну строку, то

вместо ее выделения достаточно установить на ней клавиатурный курсор, после чего нажать указанные клавиши. Вместо клавиатурных комбинаций можно использовать кнопки быстрого доступа  на панели **Text Editor**.

5. Редактор среды Visual C# обладает богатыми возможностями *поиска и замены*. Эти возможности подробно описываются в комментарии 2 к разд. 10.7.
6. Наконец, следует упомянуть о возможности *автогенерации кода* с помощью специальных шаблонов (code snippets). Использование подобных шаблонов описывается в разд. 31.1 и комментарии к нему.

1.6. Запуск приложения

Каждый этап разработки проекта описывается в отдельном разделе главы, посвященной этому проекту. В любом разделе вначале перечисляются необходимые действия, связанные с модификацией проекта. Затем следует абзац, начинающийся во слова **Результат**. В этом абзаце описывается, как будет работать новый вариант программы. Появление абзаца **Результат** служит признаком того, что модифицированный проект можно откомпилировать и запустить на выполнение (для этого достаточно нажать клавишу <F5> или кнопку ).

Если код программы был набран с синтаксическими ошибками, то сообщения об этих ошибках появляются в окне **Error List**, которое при этом становится активным. Для того чтобы перейти на строку программы, в которой обнаружена первая синтаксическая ошибка, достаточно нажать клавишу <↓> (в результате сообщение о первой ошибке будет подсвечено), после чего нажать <Enter>. Можно также выполнить двойной щелчок мышью на строке с сообщением об ошибке.

При компиляции программы в среде Visual C# важную роль играют две настройки среды, расположенные в группе **Projects and Solutions** окна **Options** (данное окно вызывается командой меню **Tools | Options…**). Первая настройка — флажок **Always show Error List if build finishes with errors** (Всегда отображать окно **Error List**, если компиляция завершилась с ошибками) в разделе **General**. Этот флажок должен быть установлен. Вторая настройка — выпадающий список **On Run, when build or deployment errors occur** (Если обнаружены ошибки компиляции или размещения при выполнении команды **Run**) в разделе **Build and Run**. В этом списке должен быть выбран вариант **Do not launch** (Не запускать программу). В экспресс-варианте среды Visual C# для отображения указанных настроек необходимо установить флажок **Show all settings** в окне **Options** (по умолчанию этот флажок не установлен).

Если после абзаца **Результат** следует абзац с пометкой **Ошибка**, значит, программа, несмотря на успешную компиляцию, будет работать не совсем правильно, и в нее надо внести дополнительные исправления (таким способом в примерах привлекается внимание к типичным ошибкам, которые могут возникнуть в аналогичных ситуациях). Если после абзаца **Результат** следует абзац с пометкой **Недочет**, значит, программа работает правильно и делает то, что требуется, но имеет дефекты интерфейса, то есть ею неудобно пользоваться. Как правило, сразу после описания ошибки или недочета указывается способ их исправления, хотя иногда исправление откладывается до следующего раздела.

Глава 2. Консольное приложение: проект DISKINFO

2.1. Создание консольного приложения

При создании нового проекта — консольного приложения следует выполнять практически те же действия, что и при создании Windows-приложения (см. разд. 1.1). Единственным отличием является указание другого типа проекта в окне **New Project**: для консольного приложения следует выбрать вариант **Console Application**.

Созданный проект будет содержать файл Program.cs, который сразу загрузится в редактор (листинг 2.1).

Листинг 2.1. Исходный текст файла Program.cs для консольного приложения

```
using System;
using System.Collections.Generic;
using System.Text;

namespace DISKINFO
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

Первые три оператора в файле Program.cs содержат названия трех пространств имен, классы из которых используются наиболее часто. Благодаря этим операторам, имена классов можно указывать в программе без уточнения, к какому пространству имен они принадлежат (например, вместо `System.Console` можно писать просто `Console`).

Метод `Main` является стартовой точкой выполнения программы. Его параметр `args` позволяет получить информацию о *параметрах командной строки*, указанных при запуске данной программы (использование параметра `args` демонстрируется в разд. 2.3).

Добавьте в метод `Main` новые операторы (листинг 2.2):

Листинг 2.2. Добавление к методу `Main`

```
Console.WriteLine("Программа DISKINFO\n");
Console.WriteLine("\nДля завершения программы нажмите <Enter>...");
Console.ReadLine();
```

Результат: при запуске программы на экране появляется особое *консольное окно*, используемое для ввода–вывода данных в текстовом режиме. Это окно обладает всеми свойствами окна приложения DOS; в частности, оно имеет расширенное системное меню, вызываемое комбинацией <Alt>+<пробел>. При выполнении программы можно переключаться в полноэкранный режим комбинацией <Alt>+<Enter> (эта же комбинация восстанавливает оконный режим). Консольное окно будет содержать текст, приведенный в листинге 2.3.

После завершения программы консольное окно немедленно закрывается. Чтобы можно было ознакомиться с содержимым окна, в программу был добавлен вызов метода `ReadLine` класса `Concole` (см. листинг 2.2, последний оператор). Этот метод предназначен для ввода строки, причем признаком завершения ввода является нажатие клавиши <Enter>. Поэтому до нажатия <Enter> консольное окно будет оставаться на экране. Строка, возвращаемая методом `ReadLine`, в программе не используется, поэтому сохранять ее в какой-либо переменной нет необходимости.

Листинг 2.3. Содержимое консольного окна при выполнении программы

```
Программа DISKINFO


Для завершения программы нажмите <Enter>...
```

2.2. Получение информации о текущем диске

В начало файла Program.cs добавьте оператор

```
using System.IO;
```

В класс `Program` добавьте новый метод `DInfo` (листинг 2.4) и измените метод `Main` (листинг 2.5).

Листинг 2.4. Метод `DInfo` класса `Program`

```
static void DInfo(string path)
{
  string none = "---",
    d = path[0].ToString().ToUpper();
  DriveInfo di = new DriveInfo(d);
  StringBuilder s = new StringBuilder(40);
  s.AppendFormat(" {0,-4}", d);
  if (di.DriveType != DriveType.NoRootDirectory)
  {
    s.AppendFormat(" {0,-9}", di.DriveType);
    if (di.IsReady)
      s.AppendFormat("{0,12:N0} {1,12:N0}", di.TotalSize / 1024,
        di.TotalFreeSpace / 1024);
    else
      s.AppendFormat("{0,12} {0,12}", none);
  }
  else
    s.AppendFormat(" {0,-9}{0,12} {0,12}", none);
  Console.WriteLine(s);
}
```

Листинг 2.5. Новый вариант метода `Main`

```
static void Main(string[] args)
{
  Console.WriteLine("Программа DISKINFO\n");
  Console.WriteLine(" Disk Type         Size (K)     Free (K)");
  Console.WriteLine(new String('=', 40));
  DInfo(Environment.CurrentDirectory);
  Console.WriteLine("\nДля завершения программы нажмите <Enter>...");
  Console.ReadLine();
}
```

Результат: при запуске программы в консольном окне отображается информация о текущем диске (листинг 2.6).

Листинг 2.6. Содержимое консольного окна при выполнении программы

```
Программа DISKINFO

 Disk Type         Size (K)     Free (K)
========================================
 D    Fixed      16 370 256    4 292 728

Для завершения программы нажмите <Enter>...
```

2.3. Использование параметров командной строки

Измените метод `Main` в файле Program.cs (листинг 2.7).

Листинг 2.7. Новый вариант метода `Main`

```
static void Main(string[] args)
{
  Console.WriteLine("Программа DISKINFO\n");
  Console.WriteLine(" Disk Type         Size (K)      Free (K)");
  Console.WriteLine(new String('=', 40));
  if (args.Length == 0)
    DInfo(Environment.CurrentDirectory[0]);
  else
    foreach (string d in args)
      DInfo(d);
  Console.WriteLine("\nДля завершения программы нажмите <Enter>...");
  Console.ReadLine();
}
```

Результат: если в качестве параметров командной строки указано одно или несколько имен дисков, то выводится информация об этих дисках (параметры командной строки должны отделяться друг от друга пробелами). Если параметры не указаны, то выводится информация о текущем диске. Для задания параметров командной строки в среде Visual C# следует выполнить команду **Project | DISKINFO Properties…**, в загруженном в редактор списке свойств проекта выбрать раздел **Debug** и ввести параметры командной строки в поле **Command line arguments**. Так, если в строке параметров указать **a d f z**, то в результате выполнения программы будет выведен текст, подобный приведенному в листинге 2.8.

В среде Windows программу с параметрами можно запустить, например, из меню **Пуск** командой **Выполнить...**:

```
C:\CSProjects\DISKINFO\DISKINFO.exe  a  d  f  z
```

Если программа запускается с помощью *ярлыка*, то ее параметры командной строки можно задать в окне свойств ярлыка, которое вызывается командой **Свойства** контекстного меню. В окне свойств надо перейти на вкладку **Ярлык** и указать нужные параметры в поле **Файл** (или **Объект**).

Листинг 2.8. Содержимое консольного окна при выполнении программы

```
Программа DISKINFO

 Disk Type         Size (K)      Free (K)
========================================
 A    Removable         ---           ---
 D    Fixed      16 370 256     4 285 224
 F    Fixed       6 309 192       543 420
 Z    ---               ---           ---

Для завершения программы нажмите <Enter>...
```

Недочет: если в качестве одного из параметров командной строки указать строку, начинающуюся не с латинской буквы, то при выполнении программы произойдет ошибка.

Исправление: в методе `DInfo` перед оператором

```
DiveInfo di = new DriveInfo(d);
```

добавьте следующий фрагмент:

```
if (d[0] < 'A' || d[0] > 'Z')
  return;
```

Результат: теперь программа не обрабатывает параметры, началом которых не является заглавная латинская буква.

Глава 3. Обработка исключений: проект EXCEP

3.1. Обработка конкретного исключения и групп исключений

В этом примере, как и в предыдущем, будет разрабатываться консольное приложение. Создайте заготовку проекта для консольного приложения (как в разд. 2.1) и измените описание класса `Program` в файле Program.cs (листинг 3.1).

Листинг 3.1. Описание класса `Program`

```
class Program
{
  static void M1(int x, int y, int z)
  {
    try
    {
      int a = checked((int)Math.Pow(x, y));
      Console.WriteLine("x ^ y / z = {0}", a / z);
    }
    catch (DivideByZeroException)
    {
      Console.WriteLine("DivideByZero Exception");
    }
    Console.WriteLine("M1 finished");
  }
```

```
  static void M2(int x, int y, int z)
  {
    try
    {
      M1(x, y, z);
    }
    catch (ArithmeticException)
    {
      Console.WriteLine("Arithmetic Exception");
    }
    Console.WriteLine("M2 finished");
  }
  static void Main(string[] args)
  {
    Console.Write("x = ");
    int x = int.Parse(Console.ReadLine());
    Console.Write("y = ");
    int y = int.Parse(Console.ReadLine());
    Console.Write("z = ");
    int z = int.Parse(Console.ReadLine());
    M2(x, y, z);
    Console.ReadLine();
  }
}
```

Результат: для трех введенных целых чисел *x*, *y*, *z* программа вычисляет выражение x^y/z, обрабатывая возникающие при этом *исключительные ситуации* (для выхода из программы надо нажать <Enter>). Опишем различные варианты работы программы.

Вариант A. Обработка допустимых значений (листинг 3.2). Вычисления выполняются успешно; ни один обработчик исключений не активизируется.

Листинг 3.2. Содержимое консольного окна при обработке допустимых значений

```
x = 9
y = 2
z = 3
x ^ y / z = 27
M1 finished
M2 finished
```

Вариант B. Деление на 0 (листинг 3.3). Активизируется обработчик `try`-блока метода `M1`, обрабатывающий исключение типа `DivideByZeroException`, после чего выполнение программы продолжается с оператора, следующего за данным `try`-блоком.

Листинг 3.3. Содержимое консольного окна при делении на 0

```
x = 1
y = 1
z = 0
DivideByZero Exception
M1 finished
M2 finished
```

Вариант C. Целочисленное переполнение (листинг 3.4). При попытке возведения числа 10 в степень 10 (и последующего преобразования результата к целому типу) возникает исключение `OverflowException`. Поскольку в разделе `catch` `try`-блока метода `M1` обработка исключения `OverflowException` не предусмотрена, происходит немедленный переход в обработчик `try`-блока следующего уровня (то есть в раздел `catch` `try`-блока метода `M2`). Здесь исключение `OverflowException` обрабатывается, так как оно является *потомком* исключения `ArithmeticException` — предка всех исключений, порожденных ошибками при выполнении арифметических операций. После этой обработки выполнение программы продолжается с оператора, следующего за `try`-блоком метода `M2`.

Листинг 3.4. Содержимое консольного окна при целочисленном переполнении

```
x = 10
y = 10
z = 1
Arithmetic Exception
M2 finished
```

D. Ввод недопустимого символа. После ввода недопустимого символа (например, звездочки `*`) выполнение программы *немедленно прерывается,* происходит возврат в среду Visual C# и в тексте программы выделяется оператор, выполнение которого привело к возникновению исключения (в нашем случае это будет первый из операторов метода `Main`, содержащих вызов функции `Parse`). Такое поведение программы объясняется тем, что в методе `Main` не предусмотрена обработка возникшего исключения `FormatException`, и поэтому активизируется режим обработки исключения по умолчанию.

В данной ситуации возможны два варианта действий:

1. Немедленно прервать выполнение программы, нажав комбинацию <Shift>+<F5> или кнопку с изображением синего квадратика  .
2. Продолжить выполнение программы, пропустив ошибочный оператор и, возможно, несколько следующих операторов. Для этого надо зацепить мышью желтую стрелку  , расположенную рядом с ошибочным оператором, и перетащить ее к тому оператору, с которого надо продолжить выполнение программы, после чего нажать клавишу <F5> или кнопку с изображением зеленого треугольника  . Можно также перейти к *пошаговому выполнению программы*, нажимая кнопки  (или клавишу <F11>) и  (или клавишу <F10>). Любая из этих кнопок обеспечивает выполнение текущего оператора (то есть оператора, на который указывает желтая стрелка). Отличие между ними состоит в том, что если текущий оператор является вызовом функции и код этой функции доступен, то первая кнопка (кнопка **Step Into**) обеспечивает переход на начало этой функции, а вторая (кнопка **Step Over**) немедленно выполняет функцию и переходит к следующему за ней оператору.

3.2. Обработка любого исключения

Измените метод `Main` в файле Program.cs (листинг 3.5). Теперь программа содержит три вложенных `try`-блока.

Листинг 3.5. Новый вариант метода `Main`

```
static void Main(string[] args)
{
  try
  {
    Console.Write("x = ");
    int x = int.Parse(Console.ReadLine());
    Console.Write("y = ");
    int y = int.Parse(Console.ReadLine());
    Console.Write("z = ");
    int z = int.Parse(Console.ReadLine());
    M2(x, y, z);
  }
  catch
  {
    Console.WriteLine("Other exception");
  }
  Console.ReadLine();
}
```

Результат: в любом из вариантов A, B, C, рассмотренных в разд. 3.1, результат работы программы будет тем же самым. В варианте D (ввод недопустимого символа) в окне будет выведено

```
x = *
Other exception
```

и программа будет ожидать нажатия <Enter> для своего завершения. Таким образом, никакое исключение теперь не приведет к аварийному завершению программы.

Недочет: информация, выводимая на экран, не позволяет определить, какое именно исключение возникло в ходе выполнения программы.

Исправление: измените раздел `catch` в методе `Main` на следующий:

```
catch(Exception ex)
{
  Console.WriteLine(ex.GetType().Name + ":\n  " + ex.Message);
}
```

Результат: теперь в случае ввода недопустимого символа будет выведена более содержательная информация:

```
x = *
FormatException:
  Input string was not in a correct format.
```

3.3. Повторное возбуждение обработанного исключения

Дополните раздел `catch` для `try`-блока метода `M2`:

```
catch (ArithmeticException)
{
  Console.WriteLine("Arithmetic Exception");
  throw;
}
```

Результат: в любом из вариантов A, B, D результат работы программы будет тем же самым. В варианте C (целочисленное переполнение) в окне будет выведена более содержательная информация (листинг 3.6). Это связано с тем, что добавленный в раздел `catch` метода `M2` оператор `throw` выполнил *повторное возбуждение* только что обработанного исключения. Повторно возбужденное исключение было окончательно обработано в разделе `catch` метода `Main` (см. разд. 3.2).

Листинг 3.6. Содержимое консольного окна при целочисленном переполнении

```
x = 10
```

```
y = 10
z = 1
Arithmetic Exception
OverflowException:
  Arithmetic operation resulted in an overflow.
```

Глава 4. События: проект EVENTS

4.1. Связывание события с обработчиком

Проект EVENTS является первым графическим приложением, рассматриваемым в книге, поэтому действия по его разработке мы опишем более подробно (см. также разд. 1).

После создания нового проекта типа **Windows Application** разместите на форме `Form1` компонент-кнопку типа `Button`, используя *окно компонентов* **Toolbox** (проще всего выбрать компонент **Button** из группы **All Windows Forms**, содержащей все компоненты в алфавитном порядке). Добавленной кнопке будет автоматически присвоено имя `button1`.

Настройте свойства формы `Form1` и кнопки `button1` (листинг 4.1). Для этого надо использовать *окно свойств* **Properties**.

По рис. 4.1 настройте размеры формы и расположение кнопки.

С событием `Click` компонента `button1` свяжите обработчик (листинг 4.2). Для этого надо выделить на форме кнопку **button1**, например, щелкнув на кнопке мышью (в результате вокруг кнопки будут изображены маркеры, как на рис. 4.1, а окно свойств **Properties** будет настроено на отображение свойств кнопки). Затем надо выбрать в окне **Properties** режим **Events** (нажав на кнопку с изображением молнии ) и выполнить двойной щелчок мышью на пустом поле ввода справа от метки **Click**. В результате в редактор среды Visual C# будет загружен файл Form1.cs с описанием класса `Form1`, и в этот файл будет добавлена заготовка для обработчика события `Click` (метод `button1_Click`). Теперь, используя редактор, надо ввести в эту заготовку нужные операторы (в данном случае — вызов метода `Close`). Текст метода `button1_Click`, создаваемый автоматически, изображается в листинге обычным шрифтом, а текст, который надо добавить в метод, — полужирным.

Аналогичными действиями создайте обработчик события `MouseDown` для формы `Form1` (листинг 4.3). Поскольку это событие должно быть связано не с кнопкой, а с формой, предварительно надо выделить форму `Form1`, выполнив щелчок мышью в любой ее свободной области или на ее заголовке.

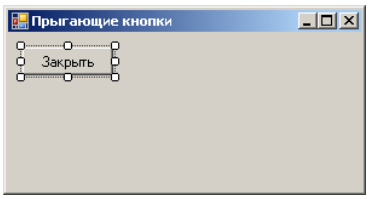


Рис. 4.1. Вид формы Form1 для проекта EVENTS на начальном этапе разработки

Листинг 4.1. Настройка свойств

```
Form1: Text = Прыгающие кнопки,
  StartPosition = CenterScreen
button1: Text = Закрыть
```

Листинг 4.2. Обработчик `button1.Click`

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

Листинг 4.3. Обработчик `Form1.MouseDown`

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    button1.Location = new Point(e.X - button1.Width / 2,
        e.Y - button1.Height / 2);
}
```

Результат: после запуска программы на экране появляется окно **Прыгающие кнопки** с кнопкой **Закрыть**. При щелчке на любом месте окна кнопка услужливо прыгает на указанное место. Благодаря значению **`CenterScreen`**, установленному для свойства `StartPosition`, окно располагается в центре экрана. Нажатие на кнопку **Закрыть** приводит к завершению программы и возврату в среду Visual C#.

4.2. Отключение обработчика от события

Добавьте на форму еще одну кнопку (она получит имя `button2`) и сделайте ее свойство `Text` пустым, используя окно **Properties** (рис. 4.2).

В файле Form1.cs в начало описания класса `Form1` (перед конструктором `public Form1()`) добавьте следующее описание объекта `r`:

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

Определите обработчики событий `MouseDown` и `Click` для кнопки `button2` (листинг 4.4).

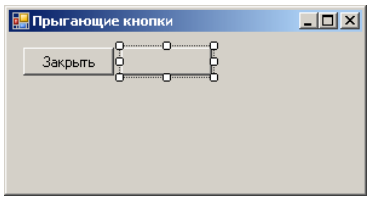


Рис. 4.2. Окончательный вид формы Form1 для проекта EVENTS

Листинг 4.4. Обработчики `button2.MouseMove` и `button2.Click`

```
private void button2_MouseMove(object sender, MouseEventArgs e)
{
    if (Control.ModifierKeys == Keys.Control)
        return;
    // - если нажата клавиша <Ctrl>,
    // то немедленно выйти из обработчика
    button2.Location = new Point(r.Next(ClientRectangle.Width - 5),
        r.Next(ClientRectangle.Height - 5));
}
private void button2_Click(object sender, EventArgs e)
{
    button2.Text = "Изменить";
    button2.MouseMove -= button2_MouseMove;
}
```

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

Прирученная кнопка пока ничего не делает. Это будет исправлено в разд. 4.3.

4.3. Подключение к событию другого обработчика

Для того чтобы прирученная кнопка при нажатии на нее выполняла какие-либо действия, можно добавить требуемые действия к уже имеющемуся обработчику `button2_Click`. Однако в этом случае обработчик должен проверять, в каком состоянии находится кнопка: "диком" или "прирученном". Поступим по-другому: свяжем событие `Click` для "прирученной" кнопки с другим обработчиком. Такой подход позволит продемонстрировать ряд особенностей, связанных с действиями по подключению и отключению обработчиков.

Новый обработчик `button2_Click2` создайте "вручную", не прибегая к услугам окна свойств **Properties**. Для этого в конце описания класса `Form1` в файле Form1.cs (*перед* двумя последними закрывающими фигурными скобками `}`) добавьте описание нового обработчика (листинг 4.5). Обратите внимание на то, что в листинге 4.5 выделены полужирным шрифтом все строки. Это означает, что необходимо набрать весь его текст. Кроме того, добавьте новые операторы в методы `button2_Click` (листинг 4.6) и `Form1_MouseDown` (листинг 4.7). Напомним, что если место добавления не уточняется, то операторы надо добавлять в *конец* метода.

Листинг 4.5. Метод `button2_Click2`

```
private void button2_Click2(object sender, EventArgs e)
{
    if (WindowState == FormWindowState.Normal)
        WindowState = FormWindowState.Maximized;
    else
        WindowState = FormWindowState.Normal;
}
```

Листинг 4.6. Добавление к методу `button2_Click`

```
button2.Click -= button2_Click;
button2.Click += button2_Click2;
```

Листинг 4.7. Добавление к методу `Form1_MouseDown`

```
if (button2.Text != "")
{
    button2.Text = "";
    button2.MouseMove += button2_MouseMove;
    button2.Click += button2_Click;
    button2.Click -= button2_Click2;
}
```

Результат: прирученная кнопка теперь выполняет полезную работу: щелчок на ней приводит к разворачиванию окна программы на весь экран, а новый щелчок восстанавливает первоначальное состояние окна. Если же щелкнуть мышью на форме (не на кнопке), то услужливая кнопка **Закрыть** прибежит на вызов, а прирученная кнопка **Изменить** снова одичает, потеряет текст своего заголовка и начнет убегать от мыши.

Недочет: при выполнении программы может возникнуть ситуация, когда одна или обе кнопки не будут отображаться на форме (если, например, кнопки были перемещены на новое место при развернутом окне, после чего окно было возвращено в исходное состояние).

Исправление: определите обработчик события `SizeChanged` для формы `Form1` (листинг 4.8).

Листинг 4.8. Обработчик `Form1.SizeChanged`

```
private void Form1_SizeChanged(object sender, EventArgs e)
{
    if (!ClientRectangle.IntersectsWith
            (button1.Bounds))
        button1.Location = new Point(10, 10);
    if (!ClientRectangle.IntersectsWith
            (button2.Bounds))
        button2.Location = new Point(10, 40);
}
```

Результат: теперь в ситуации, когда при изменении размера формы ее кнопки оказываются вне клиентской части, происходит перемещение этих кнопок на явно указанные позиции около левого верхнего угла формы.