Глава 8. Поля ввода: проект TEXTBOXES

8.1. Дополнительное выделение активного поля ввода

После создания проекта TEXTBOXES разместите в форме `Form1` 12 полей ввода `TextBox` (`textBox1`–`textBox12`) и настройте свойства формы и добавленных компонентов (листинг 8.1). Компоненты `TextBox` следует размещать на форме по строкам в направлении слева направо: `textBox1`–`textBox3` в первом ряду, `textBox4`–`textBox6` во втором ряду и т. д. (см. рис. 8.1). Для того чтобы присвоить свойству `Text` всех компонентов одно и то же значение **`Data`**, достаточно выделить все компоненты и затем задать это значение с помощью окна **Properties**.

Определите обработчики событий `Enter` и `Leave` для поля ввода `textBox1` (листинг 8.2), после чего свяжите созданные обработчики с событиями `Enter` и `Leave` всех полей ввода (о связывании обработчиков с несколькими компонентами см. разд. 6.1). Чтобы связать созданные обработчики одновременно со всеми оставшимися полями ввода (`textBox2`–`textBox12`), следует предварительно их выделить.

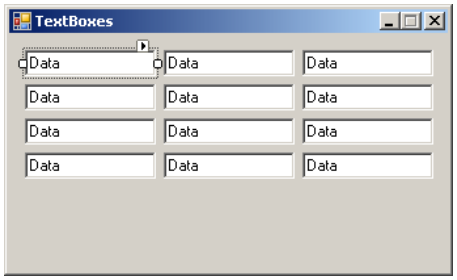

Рис. 8.1. Вид формы Form1 для проекта TEXTBOXES на начальном этапе разработки

Листинг 8.1. Настройка свойств

```
Form1: Text = TextBoxes, MaximizeBox = False,
  FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen
button1-button12: Text = Data
```

Листинг 8.2. Обработчики `textBox1.Enter` и `textBox1.Leave`

```
private void textBox1_Enter(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  tb.ForeColor = Color.White;
  tb.BackColor = Color.Green;
}
private void textBox1_Leave(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  tb.ForeColor = SystemColors.WindowText;
  tb.BackColor = SystemColors.Window;
}
```

Результат: при получении фокуса любым полем ввода (то есть при активизации поля ввода) изменяется его фон и цвет символов; при потере фокуса восстанавливается исходная цветовая настройка.

Недочет: при получении фокуса текст поля ввода выделяется (как правило, он изображается белым цветом на синем фоне); таким образом, левая часть поля ввода (выделенные символы) закрашивается синим цветом, а правая — зеленым, что выглядит некрасиво.

Исправление: добавьте новые операторы в конструктор класса `Form1` (листинг 8.3).

Листинг 8.3. Новый вариант конструктора формы `Form1`

```
public Form1()
{
  InitializeComponent();
  for (int i = 1; i <= 12; i++)
  {
    TextBox tb = Controls["textBox" + i] as TextBox;
    tb.Select(tb.Text.Length, 0);
  }
}
```

Результат: теперь при получении фокуса текст в поле ввода не выделяется, причем клавиатурный курсор (*каретка*, caret), имеющий вид вертикальной линии, располагается за последним символом текста.

8.2. Управление порядком обхода полей в форме

Разместите в форме `Form1` компонент-*контейнер* `groupBox1` (типа `GroupBox`). После этого разместите в созданном компоненте-контейнере две радиокнопки: `radioButton1` и `radioButton2`. Настройте свойства добавленных компонентов (листинг 8.4) и их положение (рис. 8.2).

Определите обработчик события `CheckedChanged` для радиокнопки `radioButton1` (листинг 8.5), после чего свяжите созданный обработчик с событием `CheckedChanged` радиокнопки `radioButton2`.

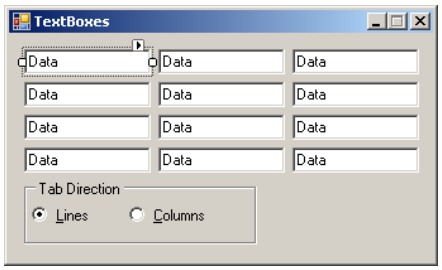

Рис. 8.2. Вид формы Form1 для проекта TEXTBOXES на промежуточном этапе разработки

Листинг 8.4. Настройка свойств

```
groupBox1: Text = Tab Direction
radioButton1: Text = &Lines, Checked = True
radioButton2: Text = &Columns
```

Листинг 8.5. Обработчик `radioButton1.Click`

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
  if (!(sender as RadioButton).Checked)
    return;
  if (sender == radioButton1)
    for (int i = 0; i <= 11; i++)
      Controls["textBox" + (i + 1)].TabIndex = i;
  else
    for (int i = 0; i <= 3; i++)
      for (int j = 0; j <= 2; j++)
        Controls["textBox" + (3 * i + j + 1)].TabIndex = i + 4 * j;
}
```

Результат: с помощью добавленных на форму радиокнопок можно изменять порядок обхода полей ввода по нажатию клавиш <Tab> и <Shift>+<Tab>: поля теперь можно обходить либо по строкам (при выбранной радиокнопке **Lines**), либо по столбцам (при выбранной радиокнопке **Columns**). Переключать порядок обхода можно также с помощью клавиатуры, используя комбинации клавиш <Alt>+<L> и <Alt>+<C>.

Недочет: при любом из реализованных способов изменения порядка обхода полей текущее поле ввода теряет фокус (поскольку фокус принимает одна из радиокнопок).

Исправление: измените свойства `Text` радиокнопок следующим образом: **`&Lines (F2)`** для `radioButton1` и **`&Columns (F3)`** для `radioButton2`, присвойте свойству `KeyPreview` формы `Form1` значение **`True`** и определите обработчик события `KeyDown` для формы `Form1` (листинг 8.6).

Листинг 8.6. Обработчик `Form1.KeyDown`

```
private void Form1_KeyDown(object sender, KeyEventArgs e)
{
  switch (e.KeyCode)
  {
    case Keys.F2:
      radioButton1.Checked = true; break;
    case Keys.F3:
      radioButton2.Checked = true; break;
  }
}
```

Результат: теперь для настройки порядка обхода полей по строкам достаточно нажать клавишу <F2>, а по столбцам — <F3>, причем текущее поле ввода сохраняет фокус.

8.3. Блокировка выхода из незаполненного поля ввода

Определите обработчик события `Validating` для поля ввода `textBox1` (листинг 8.7), после чего свяжите созданный обработчик с событиями `Validating` всех полей ввода.

Листинг 8.7. Обработчик `textBox1.Validating`

```
private void textBox1_Validating(object sender, CancelEventArgs e)
```

```
{
  e.Cancel = (sender as TextBox).Text.Trim() == "";
}
```

Результат: если активное поле ввода является пустым, то выйти из него невозможно (в частности, невозможно закрыть форму). Заметим, что при этом сохраняется возможность выбора радиокнопок клавишами <F2> и <F3>, так как эта возможность не связана с потерей фокуса для активного поля ввода.

Недочет: причина, по которой происходит блокировка фокуса, может быть непонятна пользователю. Этот недочет будет исправлен в следующем разделе.

8.4. Информирование пользователя о возникшей ошибке

Добавьте к форме невизуальный компонент `ErrorProvider` (он получит имя `errorProvider1`) и положите его свойство `BlinkStyle` равным **`NeverBlink`**.

Определите обработчик события `TextChanged` для поля ввода `textBox1` (листинг 8.8), после чего свяжите созданный обработчик с событиями `TextChanged` всех полей ввода.

Листинг 8.8. Обработчик `textBox1.TextChanged`

```
private void textBox9_TextChanged(object sender, EventArgs e)
{
  TextBox tb = sender as TextBox;
  if (tb.Text == "")
    errorProvider1.SetError(tb, "Text must be non-empty");
  else
    if (errorProvider1.GetError(tb) != "")
      errorProvider1.SetError(tb, "");
}
```

Результат: если в активном поле ввода удалить все символы, то справа от него появится значок в виде красного кружка с восклицательным знаком — признак сделанной ошибки. Если навести курсор мыши на этот значок, то появится всплывающая подсказка с кратким объяснением причины ошибки. При вводе в пустое поле хотя бы одного символа значок исчезнет.

8.5. Предоставление дополнительной информации об ошибке

Краткое сообщение об ошибке для некоторых пользователей может оказаться непонятным. В подобной ситуации желательно предусмотреть возможность вызова справочной системы, однако этот вызов, как правило, выполняется в помощью кнопки **Help**, в то время как выход из ошибочного поля ввода в программе заблокирован. Для решения этой проблемы предусмотрено свойство `CausesValidation`.

Разместите в форме кнопку `button1`, присвойте ее свойству `Text` значение **`Help`** (рис. 8.3), а свойству `CausesValidation` значение **`False`** и определите обработчик события `Click` для этой кнопки (листинг 8.9).

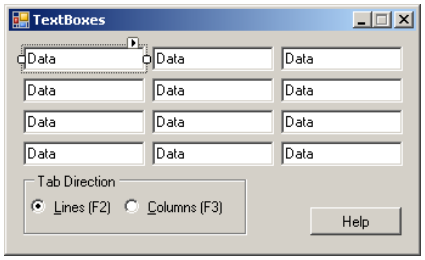

Рис. 8.3. Окончательный вид формы Form1 для проекта TEXTBOXES

Листинг 8.9. Обработчик `button1.Click`

```
private void button1_Click(object sender, EventArgs e)
{
  MessageBox.Show("Текст в поле ввода не должен быть пустым", "Help");
}
```

Результат: кнопку **Help** можно нажать мышью и в ситуации, когда одно из полей ввода заблокировано. Однако в этой ситуации по-прежнему нельзя перейти на другие компоненты формы (можно лишь вернуться на поле ввода, помеченное как ошибочное).

8.6. Проверка ошибок на уровне формы

Блокировка ошибочного поля ввода может оказаться излишне жесткой мерой для пользователей, которые предпочитают вначале заполнить поля, не вызывающие проблем, а затем вернуться к тем полям, заполнение которых требует дополнительных размышлений. При таком способе заполнения данных следует блокировать действия, выполняемые "на уровне формы", то есть на уровне всего набора данных как единого целого (например, сохранение всего набора данных в файле или пересылка этого набора по сети). Реализуем подобный вариант проверки для нашего примера.

Выделите все компоненты `TextBox` и очистите для них событие `Validating` в окне свойств **Properties**, удалите описание метода `textBox1_Validating` из файла Form1.cs и определите обработчик события `FormClosing` для формы `Form1` (листинг 8.10).

Листинг 8.10. Обработчик `Form1.FormClosing`

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
  if (e.CloseReason != CloseReason.UserClosing)
    return;
  for (int i = 1; i <=12; i++)
    if (errorProvider1.GetError(Controls["textBox" + i]) != "")
    {
      e.Cancel = true;
      return;
    }
}
```

Результат: теперь наличие пустого поля ввода не препятствует переходу в другие поля, однако около каждого пустого поля ввода изображается значок ошибки. При наличии хотя бы одного значка ошибки форму нельзя закрыть.

Глава 10. Обработка событий от мыши: проект MOUSE

10.1. Перетаскивание с помощью мыши. Настройка *z*-порядка компонентов на форме

После создания проекта MOUSE разместите в форме `Form1` два компонента типа `Panel` (они получат имена `panel1` и `panel2`) и настройте свойства формы и добавленных компонентов (листинг 10.1, рис. 10.1).

В начало описания класса `Form1` добавьте описание поля:

```
private Point p;
```

Определите обработчики событий `MouseDown` и `MouseMove` для панели `panel1` (листинг 10.2), после чего свяжите созданные обработчики с событиями `MouseDown` и `MouseMove` панели `panel2` (о связывании обработчиков с несколькими компонентами см. разд. 6.1). Кроме того, определите обработчик события `MouseMove` для формы `Form1` (листинг 10.3).

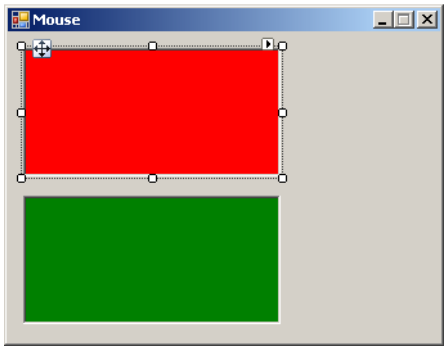

Рис. 10.1. Вид формы Form1 для проекта MOUSE на начальном этапе разработки

Листинг 10.1. Настройка свойств

```
Form1: Text = Mouse, MaximizeBox = False,
  FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen
panel1: BackColor = Red, BorderStyle = Fixed3D
panel2: BackColor = Green, BorderStyle = Fixed3D
```

Листинг 10.2. Обработчики `panel1.MouseDown` и `panel1.MouseMove`

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
}
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
    a.Location += s0;
}
```

Листинг 10.3. Обработчик `Form1.MouseMove`

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
```

```
  Text = "Mouse";
}
```

Результат: после запуска приложения перемещение курсора мыши над любой панелью приводит к тому, что в заголовке окна, кроме текста **Mouse**, выводится имя панели и текущие значения *локальных* координат мыши относительно панели. Если при этом удерживается нажатой левая кнопка мыши, то панель перемещается по форме ("перетаскивается" мышью).

Недочет: панель `panel1` при перетаскивании может заслоняться панелью `panel2`.

Исправление: добавьте в метод `panel1_MouseDown` два оператора:

```
Panel a = sender as Panel;
a.BringToFront();
```

Результат: теперь перетаскиваемая панель всегда располагается поверх остальных компонентов окна.

10.2. Изменение размеров с помощью мыши

В начало описания класса `Form1` добавьте описание нового поля:

```
private Size s;
```

Добавьте новые операторы в методы `panel1_MouseDown` и `panel1_MouseMove` (листинг 10.4).

Листинг 10.4. Новые варианты методов `panel1_MouseDown` и `panel1_MouseMove`

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
}
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
    a.Location += s0;
  else
    if (e.Button == MouseButtons.Right)
      a.Size = s + s0;
}
```

Результат: если перемещать курсор мыши над панелью, удерживая нажатой правую кнопку мыши, то происходит изменение размеров панели (левая кнопка по-прежнему используется для изменения положения панели). Следует обратить внимание на то, что перетащить панель можно даже на область вне окна. Если при этом не отпускать кнопку мыши, то панель можно вернуть обратно на видимую часть окна (аналогично, уменьшив размеры панели до нулевых, можно затем восстановить их). Отмеченная особенность связана с явлением *захвата мыши* (mouse capture): если нажать над компонентом какую-либо кнопку мыши, то этот компонент захватит мышь и "заставит" ее передавать ему все сообщения (даже если курсор мыши покинет компонент) до тех пор, пока кнопка мыши не будет отпущена (впрочем, это событие тоже будет обработано компонентом, ранее захватившим мышь).

Недочет: если панель перетащить целиком за границы окна и *отпустить* кнопку мыши, то доступ к панели окажется невозможным. Недоступной панель станет и при уменьшении ее размеров до нулевых, если при этом отпустить кнопку мыши.

Исправление: установите свойство `AutoScroll` формы `Form1` равным **`True`** и измените метод `panel1_MouseMove` (листинг 10.5).

Листинг 10.5. Исправленный вариант метода `panel1_MouseMove`

```
private void panel1_MouseMove(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  Text = string.Format("Mouse - {0} {1}", a.Name, e.Location);
  Size s0 = new Size(e.X - p.X, e.Y - p.Y);
  if (e.Button == MouseButtons.Left)
  {
    Point p0 = a.Location + s0;
    a.Location = new Point(Math.Max(0, p0.X), Math.Max(0, p0.Y));
  }
  else
  if (e.Button == MouseButtons.Right)
  {
    s0 += s;
    a.Size = new Size(Math.Max(50, s0.Width), Math.Max(20, s0.Height));
  }
}
```

Результат: теперь перетаскивать панель за пределы левой или верхней границы окна невозможно, и, кроме того, определен минимальный размер панели (в пикселах), равный 50 × 20. Перетаскивание панели за пределы правой или нижней границы окна также не делает ее недоступной, поскольку при этом на окне автоматически появляются полосы прокрутки (благодаря значению **True** для свойства формы AutoScroll).

10.3. Использование дополнительных курсоров

Добавьте новые операторы в метод panel1_MouseDown (листинг 10.6). Определите обработчик события MouseUp для панели panel1 (листинг 10.7), после чего свяжите созданный обработчик с событием MouseUp панели panel2

Листинг 10.6. Новый вариант метода panel1_MouseDown

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
  if (e.Button == MouseButtons.Left)
    a.Cursor = Cursors.Hand;
  else
    if (e.Button == MouseButtons.Right)
      a.Cursor = Cursors.SizeNWSE;
}
```

Листинг 10.7. Обработчик panel1.MouseUp

```
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  (sender as Panel).Cursor = null;
}
```

Результат: в режиме изменения размеров панели курсор мыши принимает вид диагональной двунаправленной стрелки, а в режиме перетаскивания — "указывающей руки". После выхода из этих режимов восстанавливается стандартный вид курсора.

Примечание

Свойство Cursor, имеющееся у любого визуального компонента, а также класс Cursors были подробно рассмотрены в гл. 7. Напомним, что если свойство Cursor равно null, то компонент будет использовать курсор своего родителя (в данном случае формы).

10.4. Обработка ситуации с одновременным нажатием двух кнопок мыши: первый вариант

Наша программа будет прекрасно работать до тех пор, пока пользователю не придет в голову мысль нажать левую кнопку мыши при нажатой правой кнопке (или наоборот). Для определенности опишем поведение программы в ситуации, когда при нажатой на панели левой кнопке мыши была дополнительно нажата правая (и, кроме того, будем пока считать, что при выполнении описываемых далее действий курсор мыши не будет выходить за пределы панели, на которой он первоначально находился). В момент нажатия второй кнопки вид курсора изменится на диагональную стрелку, однако при последующем перемещении мыши ни положение, ни размер панели изменяться не будут. Если отпустить одну из кнопок мыши, то курсор примет вид стандартной стрелки (вид курсора по умолчанию), однако при перемещении мыши будет выполняться действие, определяемое той кнопкой мыши, которая осталась нажатой.

Подобное поведение объясняется следующими особенностями обработчиков событий от мыши: при выполнении обработчиков для событий MouseDown и MouseUp параметр e.Button содержит информацию о той кнопке мыши, которая только что была нажата (или, соответственно, отпущена). Информация о прочих нажатых кнопках в параметре e.Button не содержится. С другой стороны, при выполнении обработчика события MouseMove параметр e.Button содержит информацию обо *всех* нажатых в данный момент кнопках мыши; при этом элементы перечисления MouseButtons, соответствующие нажатым кнопкам, объединяются операцией | (побитовое ИЛИ).

Последнее обстоятельство позволяет понять, почему при одновременно нажатых кнопках наша программа не выполняет никаких действий: в самом деле, действия по перемещению или изменению размера панели реализуются в обработчике события MouseMove при выполнении соответственно условий e.Button == MouseButtons.Left или e.Button == MouseButtons.Right, но если нажаты обе кнопки мыши, то ни одно из этих условий не является истинным, поскольку в этом случае свойство e.Button содержит выражение MouseButtons.Left | MouseButtons.Right.

Внесем в программу изменения, которые позволят более естественным образом обрабатывать ситуацию, связанную с одновременным нажатием левой и правой кнопок мыши (листинг 10.8).

Листинг 10.8. Новые варианты методов panel1_MouseDown и panel1_MouseUp

```
private void panel1_MouseDown(object sender, MouseEventArgs e)
{
  p = e.Location;
  Panel a = sender as Panel;
  a.BringToFront();
  s = a.Size;
  if (Control.MouseButtons != MouseButtons.Left
```

```
    && Control.MouseButtons != MouseButtons.Right)
    a.Cursor = null;
  else
    if (e.Button == MouseButtons.Left)
      a.Cursor = Cursors.Hand;
    else
      if (e.Button == MouseButtons.Right)
        a.Cursor = Cursors.SizeNWSE;
}
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  if (Control.MouseButtons == MouseButtons.Left)
    a.Cursor = Cursors.Hand;
  else
  if (Control.MouseButtons == MouseButtons.Right)
    a.Cursor = Cursors.SizeNWSE;
  else
    a.Cursor = null;
}
```

Результат: как и ранее, при одновременно нажатых левой и правой кнопках мыши никакое действие не выполняется. Однако теперь это согласуется с тем, что в данной ситуации курсор мыши имеет вид курсора по умолчанию. Далее, при отпускании одной из двух нажатых кнопок курсор принимает вид, соответствующий действию для той кнопки мыши, которая остается нажатой, и при перемещении мыши это действие выполняется.

10.5. Обработка ситуации с одновременным нажатием двух кнопок мыши: второй вариант

Исправления, сделанные в разд. 10.4, не решают всех проблем, связанных с одновременным нажатием двух кнопок мыши. Так, если при двух нажатых кнопках переместить курсор мыши в другую позицию панели, а затем отпустить одну из кнопок, то, в зависимости от того, какая кнопка осталась нажатой, произойдет скачкообразное изменение либо положения, либо размеров данной панели.

Для исправления этого недочета в очередной раз изменим метод `panel1_MouseUp` (листинг 10.9).

Листинг 10.9. Новый вариант метода `panel1_MouseUp`

```
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  if (Control.MouseButtons == MouseButtons.Left)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Left, 0, e.X, e.Y, 0));
  else
  if (Control.MouseButtons == MouseButtons.Right)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Right, 0, e.X, e.Y, 0));
  else
    a.Cursor = null;
}
```

Результат: теперь в ситуации, описанной выше, скачкообразного изменения свойств панели не происходит.

10.6. Обработка ситуации с одновременным нажатием двух кнопок мыши: третий вариант

К сожалению, проблемы, связанные с одновременным нажатием левой и правой кнопок мыши, на этом не заканчиваются. Предположим теперь, что пользователь выполнит такую последовательность действий:

1. Вначале пользователь нажмет обе кнопки мыши над одной из панелей.
2. Затем он переместит курсор мыши на свободную часть формы (это можно сделать, так как при двух нажатых кнопках перемещение мыши не выполняет действий, связанных с изменением положения или размеров панели).
3. На этой свободной части формы пользователь отпустит одну из кнопок (для определенности, будем считать, что отпущена правая кнопка).
4. Затем пользователь опять переместит курсор мыши на панель.

После выполнения действия 3 перемещение курсора мыши по свободной части формы никак не будет влиять на положение панели (курсор будет иметь стандартный вид). Однако после выполнения действия 4 панель "прыгнет" на новое место, причем величина и направление прыжка будет зависеть от того, в каком месте формы пользователь *отпустил правую кнопку мыши*. Если в результате прыжка панель будет задвинута в левый верхний угол формы и удастся навести на нее курсор мыши, то мы увидим, что он имеет вид руки. Однако стоит его вернуть обратно на форму, как он опять примет стандартный вид.

Все эти странности объясняются тем, что режим захвата мыши завершается, как только вне компонента, захватившего мышь, будет отпущена *какая-либо* кнопка мыши. При отпускании левой кнопки мыши над формой мышь освободится от захвата панелью, однако не будет захвачена формой (поскольку для захвата надо *нажать* на кнопку). Заметим, что перед освобождением будет вызван обработчик события `MouseUp` для той панели, которая ранее захватила мышь, и в результате вызова этого обработчика (листинг 10.9) будет вызван метод `panel1_MouseDown` для этой панели. Таким образом, для

данной панели будет сохранена информация, привязанная к той точке, в которой была отпущена кнопка мыши. Так как теперь мышь не является захваченной, ее перемещение по свободной части формы обрабатывается самой формой (при этом ничего не происходит, за исключением того, что заголовок формы постоянно имеет вид **Mouse**). Но если подобная "незахваченная" мышь окажется над панелью (причем над *любой* панелью), то сработает обработчик `MouseMove` этой панели, а поскольку в этот момент левая кнопка мыши все еще является нажатой, то эта панель будет перемещена на новое место с учетом той информации, которая была сохранена в момент отпускания правой кнопки мыши на форме.

Для того чтобы программа более естественным образом реагировала на действия, описанные в пунктах 1–4, еще раз изменим метод `panel1_MouseUp` (листинг 10.10).

Листинг 10.10. Новый вариант метода `panel1_MouseUp`

```
private void panel1_MouseUp(object sender, MouseEventArgs e)
{
  Panel a = sender as Panel;
  if (Control.MouseButtons != MouseButtons.None
    && !a.ClientRectangle.Contains(e.Location))
  {
    a.Cursor = null;
    this.Capture = true;
    return;
  }
  if (Control.MouseButtons == MouseButtons.Left)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Left,
      0, e.X, e.Y, 0));
  else
  if (Control.MouseButtons == MouseButtons.Right)
    panel1_MouseDown(sender, new MouseEventArgs(MouseButtons.Right,
      0, e.X, e.Y, 0));
  else
    a.Cursor = null;
}
```

Результат: если теперь выполнить действия 1–4, то положение панели останется неизменным, а курсор мыши будет иметь стандартный вид независимо от того, где он находится.

10.7. Перетаскивание компонентов любого типа. Поиск и замена в текстах программ

Разместите в форме `Form1` кнопку `button1` и метку `label1` и настройте свойства метки (листинг 10.11). Размеры кнопки и метки настройте в соответствии с рис. 10.2.

Свяжите обработчики `panel1_MouseDown`, `panel1_MouseMove` и `panel1_MouseUp` с соответствующими событиями (`MouseDown`, `MouseMove` и `MouseUp`) для кнопки `button1` и метки `label1`.

В тексте файла Form1.cs выполните замену слова `Panel` на слово `Control`. Для этого перейдите в начало данного файла и нажмите комбинацию клавиш <Ctrl>+<H>. В появившемся окне **Find and Replace** в поле **Find what** (Найти) введите текст `Panel`, в поле **Replace with** (Заменить на) введите текст `Control` и установите флажок **Match whole word** (Искать слово целиком); если этот флажок в окне не отображается, то нажмите кнопку **Find options** (Опции поиска). После этого нажмите кнопку **Replace all** (Заменить все). После выполнения всех замен будет выведено окно с информацией о том, что заменено 6 найденных слов `Panel` (все эти слова содержатся в трех одинаковых операторах вида `Panel a = sender as Panel`, находящихся в начале методов `panel1_MouseUp`, `panel1_MouseDown` и `panel1_MouseMove`). Для закрытия окна **Find and Replace** нажмите клавишу <Esc>.

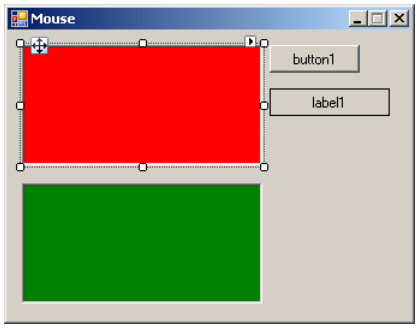

Рис. 10.2. Окончательный вид формы Form1 для проекта MOUSE

Листинг 10.11. Настройка свойств

```
label1: AutoSize = False,
  BorderStyle = FixedSingle,
  TextAlign = MiddleCenter
```

Результат: добавленные компоненты (кнопка `button1` и метка `label1`) обрабатываются так же, как и панели `panel1` и `panel2`: их размер и положение можно изменять с помощью мыши.

Комментарий

Для быстрого поиска требуемого фрагмента текста удобно использовать режим **Incremental Search**, устанавливаемый командой меню **Edit | Advanced | Incremental Search** или комбинацией <Ctrl>+<I>. При установке этого режима курсор мыши принимает вид стрелки, направленной вниз, рядом с которой изображается бинокль. Теперь достаточно напечатать требуемый текстовый фрагмент (вводимый текст отображается на статусной панели окна Visual C#), и в процессе печати в тексте файла будет выделяться первый из найденных фрагментов, расположенных ниже позиции клавиатурного курсора (то есть поиск осуществляется *вперед*). Если ввод текстового фрагмента завершен, то для поиска следующего его вхождения далее по тексту достаточно еще раз нажать <Ctrl>+<I>, а для поиска его вхождения назад по тексту — <Ctrl>+<Shift>+<I>. Для выхода из режима **Incremental Search** достаточно нажать <Esc>. Заметим, что и после выхода из данного режима редактор помнит последний введенный фрагмент для поиска, и если после следующей установки режима **Incremental Search** не набирать новый фрагмент, а сразу нажать <Ctrl>+<I> или <Ctrl>+<Shift>+<I>, то будет выполнен поиск ранее введенного фрагмента текста (в направлении вперед или назад соответственно).

Если при поиске необходимо учитывать дополнительные условия, то следует отобразить диалоговое окно **Find and Replace** в режиме **Quick Find**, нажав комбинацию <Ctrl>+<F>. Помимо уже упомянутой возможности поиска слова целиком (**Match whole word**) можно также организовать поиск с учетом регистра (**Match case**) и изменить направление поиска на обратное (**Search up**). Если установить флажок **Use** и выбрать из выпадающего списка вариант **Wildcards** (Шаблоны), то во фрагменте для поиска можно указывать следующие шаблоны:

- `*` означает один или более произвольных символов;
- `?` означает ровно один произвольный символ;
- `#` означает ровно одну цифру;
- `[`*символы*`]` означает любой символ из указанного набора;
- `[!`*символы*`]` означает любой символ, не входящий в указанный набор.

Для того чтобы любой из указанных выше символов считался не шаблоном, а обычным символом, перед ним следует указать символ \ (обратная косая черта). Обычный символ "обратная косая черта" должен указываться как `\\`. Символы-шаблоны можно быстро ввести в поле **Find what**, если воспользоваться кнопкой  , расположенной справа от этого поля.

Еще более мощные возможности предоставляет режим поиска с использованием *регулярных выражений* (**Use Regular expressions**), который здесь не рассматривается.

При открытом окне **Find and Replace** для поиска вхождения очередного фрагмента достаточно нажать кнопку **Find Next** или клавишу <Enter>. Поиск можно осуществлять и при закрытом окне, нажимая клавишу <F3>. Подчеркнем, что нажатие <F3>, в отличие от <Ctrl>+<I>, не требует ввода фрагмента и обеспечивает учет при поиске всех настроек, указанных в окне **Find and Replace**.

Еще одной полезной возможностью, ускоряющей поиск, является выпадающий список ранее искавшихся строк. Этот список находится в верхней панели инструментов, и перейти в него можно комбинацией <Ctrl>+</>. После этого можно выбрать из списка фрагмент, который искался ранее, или ввести новый текст и нажать <Enter>. В результате в редакторе будет выделено первое вхождение указанного фрагмента, расположенное далее по тексту. Повторное нажатие <Enter> выделит следующее вхождение и т. д. (все вхождения будут перебираться циклически). Для возврата в редактор достаточно нажать <Esc>.

Как уже отмечалось, окно для *поиска и замены* можно отобразить на экране с помощью комбинации <Ctrl>+<H>. Оно содержит те же флажки настроек, что и окно поиска. Шаблоны в этом окне можно указывать только в строке **Find what**. В режиме поиска и замены можно искать очередное вхождение фрагмента (кнопка **Find Next**), заменять обнаруженное вхождение (кнопка **Replace**, после выполнения замены сразу выполняется поиск следующего вхождения) или заменять все найденные вхождения (кнопка **Replace All**).

Если окно **Find and Replace** является активным, то для его закрытия достаточно нажать клавишу <Esc>.

Глава 11. Перетаскивание (drag & drop): проект ZOO

11.1. Перетаскивание меток по форме

После создания проекта ZOO разместите в форме `Form1` четыре метки (`label1`—`label4`) и настройте свойства формы и добавленных меток (листинг 11.1). Положение меток настройте в соответствии с рис. 11.1.

Определите обработчик события `MouseDown` для метки `label1` (листинг 11.2), после чего свяжите созданный обработчик с событиями `MouseDown` меток `label2`—`label4` (о связывании обработчиков с несколькими компонентами см. разд. 6.1). Кроме того, определите обработчики событий `DragEnter` и `DragDrop` для формы `Form1` (листинг 11.3).

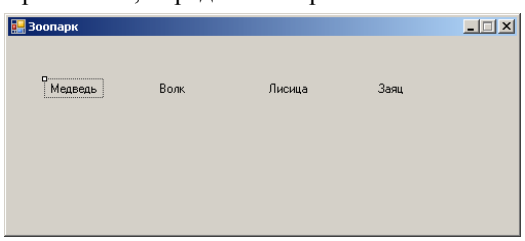

Рис. 11.1. Вид формы Form1 для проекта ZOO на начальном этапе разработки

Листинг 11.1. Настройка свойств

```
Form1: Text = Зоопарк, MaximizeBox = False,
  FormBorderStyle = FixedSingle,
  StartPosition = CenterScreen, AllowDrop = True
label1: Text = Медведь
label2: Text = Волк
label3: Text = Лисица
label4: Text = Заяц
```

Листинг 11.2. Обработчик label1.MouseDown

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    DoDragDrop(sender, DragDropEffects.Move);
}
```

Листинг 11.3. Обработчики Form1.DragEnter и Form1.DragDrop

```
private void Form1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
}
private void Form1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  src.Location = PointToClient(new Point(e.X, e.Y));
}
```

Результат: метки с названиями зверей можно перетаскивать с помощью левой кнопки мыши. При перетаскивании метки-*источника* (source) она остается на месте, однако вид курсора мыши изменяется, что является признаком режима перетаскивания. В качестве *приемника* (target) пока определена лишь сама форма: при отпускании над ней кнопки мыши происходит перемещение метки зверя на указанную позицию.

Недочет 1: разрешается перетащить метку не только на свободную часть формы, но и на другую метку; при таком перетаскивании происходит наложение двух меток.

Исправление: для всех меток установите свойство AllowDrop равным **True**.

Результат: теперь при перемещении одной метки на другую курсор мыши принимает вид запрещающего знака.

Недочет 2: в начальный момент перетаскивания курсор принимает вид запрещающего знака.

Исправление: определите обработчик события DragEnter для метки label1 (листинг 11.4), после чего свяжите созданный обработчик с событиями DragEnter меток label2–label4.

Листинг 11.4. Обработчик label1.DragEnter

```
private void label1_DragEnter(object sender, DragEventArgs e)
{
  if (e.Data.GetData(typeof(Label)) == sender)
    e.Effect = DragDropEffects.Move;
  else
    e.Effect = DragDropEffects.None;
}
```

Результат: в начальный момент перетаскивания курсор мыши остается разрешающим. Перетаскивание одной метки на другую по-прежнему запрещено.

11.2. Перетаскивание меток в поля ввода

Разместите в форме Form1 четыре поля ввода (textBox1–textBox4) и настройте их положение в соответствии с рис. 11.2. Для каждого поля ввода очистите его свойство Text и положите равными **True** свойства ReadOnly и AllowDrop (настройка ReadOnly = **True** запрещает пользователю редактировать текст в поле ввода, однако не влияет на возможность программного изменения текста).

Определите обработчики событий DragEnter и DragDrop для поля ввода textBox1 (листинг 11.5), после чего свяжите созданные обработчики с событиями DragEnter и DragDrop полей ввода textBox2–textBox4.

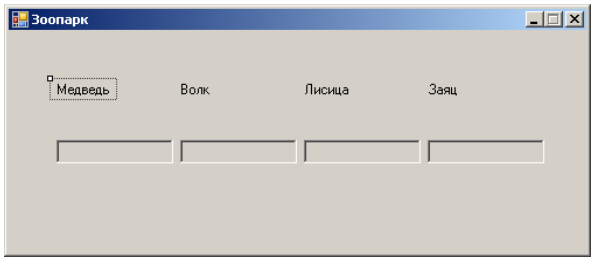

Рис. 11.2. Вид формы Form1 для проекта ZOO на промежуточном этапе разработки

Листинг 11.5. Обработчики `textBox1.DragEnter` и `textBox1.DragDrop`

```
private void textBox1_DragEnter(object sender, DragEventArgs e)
{
  if ((sender as TextBox).Text == "")
    e.Effect = DragDropEffects.Move;
  else
    e.Effect = DragDropEffects.None;
}
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  (sender as TextBox).Text = src.Text;
  src.Visible = false;
}
```

Результат: теперь приемником может также служить любое *незаполненное* поле ввода ("пустая клетка"). При перетаскивании метки на незаполненное поле ввода "зверь попадает в клетку" (текст метки отображается в поле ввода). Перетаскивание метки на уже заполненное поле ввода пока запрещено, хотя в следующем разделе это действие станет доступным.

11.3. Взаимодействие меток при их перетаскивании друг на друга

С помощью окна **Properties** установите значения свойства `Tag` для всех компонентов, размещенных в форме: `label1` — **3**, `label2` — **2**, `label3` — **1**, `label4` — **0**, `textBox1-textBox4` — **0**. После установки свойств `Tag` загрузите в редактор файл Form1.Designer.cs (достаточно выполнить двойной щелчок на его имени в окне **Solution Explorer**), разверните в этом файле раздел **Windows Form Designer generated code** и откорректируйте все 8 операторов, задающих значения свойства `Tag`, удалив в них кавычки. Например, текст

```
this.label1.Tag = "3";
```

следует заменить на

```
this.label1.Tag = 3;
```

Измените методы `label1_DragEnter` и `textBox1_DragDrop` (листинг 11.6).

Определите обработчик события `DragDrop` для метки `label1` (листинг 11.7), после чего свяжите созданный обработчик с событиями `DragDrop` меток `label2-label4`.

Свяжите обработчик `label1_DragEnter` с событиями `DragEnter` полей ввода `textBox1–textBox4`. После выполнения этого действия метод `textBox1_DragEnter` уже не будет связан ни с одним событием, поэтому его описание в файле Form1.cs можно удалить.

Листинг 11.6. Новый вариант методов `label1_DragEnter` и `textBox1_DragDrop`

```
private void label1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move
}
private void textBox1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  TextBox trg = sender as TextBox;
  if ((int)src.Tag >= (int)trg.Tag)
  {
    trg.Text = src.Text;
    trg.Tag = src.Tag;
  }
  src.Visible = false;
}
```

Листинг 11.7. Обработчик `label1.DragDrop`

```
private void label1_DragDrop(object sender, DragEventArgs e)
{
  Label src = e.Data.GetData(typeof(Label)) as Label;
  Label trg = sender as Label;
  if ((int)src.Tag > (int)trg.Tag)
  {
    src.Location = trg.Location;
    trg.Visible = false;
  }
  else
    src.Visible = false;
}
```

Результат: при перетаскивании названия одного зверя на название другого более сильный "поедает" более слабого. То же самое происходит, если один из зверей перетаскивается в клетку, уже занятую другим зверем. Заметим, что теперь события `DragEnter` всех компонентов (и меток, и полей ввода) связаны с одним и тем же обработчиком `label1_DragEnter`.

Ошибка: если при перетаскивании метки отпустить ее над ней самой, то метка исчезнет. Таким образом, зверь поедает самого себя.

Исправление: в методе `label1_DragDrop` (листинг 11.7) перед оператором

```
if ((int)src.Tag > (int)trg.Tag)
```

вставьте

```
if (src == trg) return;
```

11.4. Действия в случае перетаскивания на недопустимый приемник

Измените метод `label1_MouseDown` (листинг 11.8).

Листинг 11.8. Новый вариант метода `label1_MouseDown`

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
    if (DoDragDrop(sender, DragDropEffects.Move) == DragDropEffects.None)
      (sender as Label).Visible = false;
}
```

Результат: если перетаскивание метки-зверя завершается за пределами формы (в этом случае курсор имеет вид запрещающего знака), то зверь "убегает" из зоопарка и его метка на форме исчезает.

11.5. Дополнительное выделение источника и приемника в ходе перетаскивания

Измените методы `label1_MouseDown` и `label1_DragEnter` (листинг 11.9).

В методы `label1_DragDrop` и `textBox1_DragDrop` добавьте оператор

```
trg.BackColor = SystemColors.Control;
```

Определите обработчик события `DragLeave` для метки `label1` (листинг 11.10), после чего свяжите созданный обработчик с событиями `DragLeave` меток `label2`–`label4` и полей ввода `textBox1`–`textBox4`.

Листинг 11.9. Новый вариант методов `label1_MouseDown` и `label1_DragEnter`

```
private void label1_MouseDown(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
  {
    Label src = sender as Label;
    src.ForeColor = Color.Blue;
    if (DoDragDrop(sender, DragDropEffects.Move) == DragDropEffects.None)
      src.Visible = false;
    src.ForeColor = SystemColors.ControlText;
  }
}
private void label1_DragEnter(object sender, DragEventArgs e)
{
  e.Effect = DragDropEffects.Move;
  (sender as Control).BackColor = Color.Yellow;
}
```

Листинг 11.10. Обработчик `label1.DragLeave`

```
private void label1_DragLeave(object sender, EventArgs e)
{
  (sender as Control).BackColor = SystemColors.Control;
}
```

Результат: в режиме перетаскивания цвет текста метки-источника изменяется на синий, а текущий компонент-приемник (метка или поле ввода) изображается на желтом фоне.

Недочет: при выполнении *щелчка* на любой из меток ее фон меняется на желтый.

Исправление: в методе `label1_DragDrop` переместите только что добавленный в него оператор

```
trg.BackColor = SystemColors.Control;
```

на позицию *перед* оператором

```
if (src == trg) return;
```

Результат: теперь щелчок на метке не приводит к изменению ее внешнего вида.

11.6. Настройка вида курсора в режиме перетаскивания

Определите обработчик события `GiveFeedback` для формы `Form1` (листинг 11.11)

Листинг 11.11. Обработчик `Form1.GiveFeedback`

```
private void Form1_GiveFeedback(object sender, GiveFeedbackEventArgs e)
{
  e.UseDefaultCursors = false;
  Cursor.Current = e.Effect == DragDropEffects.Move ?
    Cursors.Hand : Cursors.No;
}
```

Результат: в режиме перетаскивания курсор мыши имеет вид "указывающей руки" (при выходе курсора за границы формы он по-прежнему принимает вид запрещающего знака).

11.7. Информация о текущем состоянии программы. Кнопки с изображениями

Разместите в форме `Form1` кнопку `button1`, в качестве ее надписи (то есть свойства `Text`) укажите **`Зоопарк закрыт`** и настройте положение кнопки в соответствии с рис. 11.3.

Для большей наглядности добавим слева от надписи кнопки небольшое изображение, вид которого (как и текст кнопки) будет зависеть от текущего состояния программы. Будем использовать рисунки из файлов Critical.bmp и OK.bmp, находящихся в коллекции изображений Visual Studio 2005 (подкаталог Common7\VS2005ImageLibrary\bitmaps\misc). Аналогичные файлы имеются в коллекции Visual Studio 2008 (подкаталог Common7\VS2008ImageLibrary\Annotations&Buttons\bmp_format). Если указанные коллекции недоступны, то можно использовать любые растровые рисунки (формата .bmp), имеющие размер 16 × 16 пикселов. Добавьте в форму компонент `ImageList`, предназначенный для хранения набора изображений одинакового размера (этому компоненту будет присвоено имя `imageList1`). Для добавления к компоненту `imageList1` рисунков Critical.bmp и OK.bmp выполните следующие действия:

- выберите в окне свойств компонента `imageList1` свойство `Images` и нажмите на кнопку с многоточием рядом с этим свойством;
- в появившемся окне **Images Collection Editor** нажмите кнопку **Add** и выберите файл **Critical.bmp** в появившемся окне открытия файла (в результате файл Critical.bmp будет добавлен в коллекцию `Images`);
- аналогичными действиями добавьте в коллекцию рисунок OK.bmp, после чего закройте окно **Images Collection Editor**, нажав клавишу <Enter> или кнопку **OK**.

Настройте остальные требуемые свойства компонентов `imageList1` и `button1` (листинг 11.12; при настройке свойства `ImageAlign` компонента `button1` надо выбрать в появившейся панели квадрат, расположенный в левом нижнем углу).

Измените метод `label1_MouseDown`, добавив к нему новые операторы (листинг 11.13).

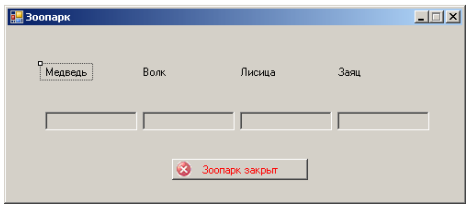

Рис. 11.3. Окончательный вид формы Form1 для проекта ZOO

Листинг 11.12. Настройка свойств

```
imageList1: TransparentColor = Magenta
button1: ForeColor = Red, ImageList = imageList1,
  ImageKey = Critical.bmp, ImageAlign = BottomLeft
```

Листинг 11.13. Добавление к методу `label1_MouseDown`

```
string s = "";
for (int i = 1; i <= 4; i++)
{
  if (Controls["label" + i].Visible)
  // один из зверей на свободе
    return;
  s = s + (Controls["textBox" + i].Text);
}
if (s == "")
  // все клетки пусты
  return;
button1.Text = "Зоопарк открыт";
button1.ForeColor = Color.Green;
button1.ImageKey = "OK.bmp";
```

Результат: если все метки на форме исчезли и при этом хотя бы одно поле ввода оказалось заполненным, то на кнопке выводится текст **Зоопарк открыт**, а цвет оформления кнопки меняется с красного на зеленый.

Недочет: перетаскивание метки на кнопку `button1` приводит к тому, что метка "проваливается" под кнопку и становится недоступной для последующего перетаскивания, поскольку ее не удается зацепить мышью.

Исправление: положите свойство `AllowDrop` кнопки `button1` равным **`True`** и свяжите с событием `DragEnter` кнопки обработчик **`Form1_DragEnter`.**

Результат: теперь при попытке перетаскивания метки на кнопку `button1` ничего не происходит: метка остается на прежнем месте.

11.8. Восстановление исходного состояния

Определите обработчики события `Load` для формы `Form1` и события `Click` для кнопки `button1` (листинг 11.14)

Листинг 11.14. Обработчик `Form1.Load`

```
private void Form1_Load(object sender, EventArgs e)
{
  for (int i = 1; i <= 4; i++)
    Controls["label" + i].Location =
      Controls["textBox" + i].Location - new Size(0, textBox1.Top / 2);
  ActiveControl = button1;
}

private void button1_Click(object sender, EventArgs e)
{
  Form1_Load(this, null);
  for (int i = 1; i <= 4; i++)
  {
    Controls["label" + i].Visible = true;
    Control c = Controls["textBox" + i];
    c.Text = "";
    c.Tag = 0;
  }
  button1.Text = "Зоопарк закрыт";
  button1.ForeColor = Color.Red;
  button1.ImageKey = "Critical.bmp";
}
```

Результат: исходное положение меток-зверей теперь определяется программно, а именно, в обработчике события формы `Load` (метки располагаются над соответствующими полями ввода и выравниваются по их левой границе). В дальнейшем при нажатии на кнопку `button1` исходное положение зверей восстанавливается, а клетки освобождаются. Кроме того, при запуске программы кнопка `button1` становится активной, что приводит к появлению вокруг нее рамки красного цвета (цвет рамки вокруг активной кнопки, определяется цветом `ForeColor` ее надписи).