Виктор Зиборов

Visual C# 2010
НА ПРИМЕРАХ

Санкт-Петербург
«БХВ-Петербург»
2011

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1
3-59

Зиборов В. В.

3-59 Visual C# 2010 на примерах. — СПб.: БХВ-Петербург, 2011. — 432 с.: ил. + CD-ROM

ISBN 978-5-9775-0698-4

Рассмотрено более 120 типичных примеров, встречающихся в практике реального программирования для платформы .NET Framework в среде Microsoft Visual C# 2010: обработка событий мыши и клавиатуры, чтение/запись файлов, редактирование графических данных, управление буфером обмена, ввод/вывод данных, использование функций MS Word, MS Excel, AutoCAD и MATLAB, использование технологий LINQ и ADO.NET при работе с базами данных, разработка интерактивных Web-приложений, создание Web-служб, разработка WPF-приложений и многое другое. Материал располагается по принципу от простого к сложному, что позволяет использовать книгу одновременно как справочник для опытных и как пособие для начинающих программистов. Компакт-диск содержит исходные коды примеров из книги.

Для программистов

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.01.11.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 34,83.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.005770.05.09
от 26.05.2009 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0698-4

Оглавление

Введение

Система разработки программного обеспечения Microsoft Visual C# 2010 является хорошим средством *быстрой разработки программ* для ускоренного создания приложений для Microsoft Windows и Интернета. Цель книги — популяризация программирования. Автор стремился показать, как *малой кровью* можно написать, почти сконструировать, *как в детском конструкторе*, довольно-таки функционально сложные приложения. Для реализации этой цели автор выбрал форму демонстрации *на примерах* решения задач от самых простых, элементарных, до более сложных.

Так, рассмотрены примеры программ с экранной формой и элементами управления в форме, такими как текстовое поле, метка, кнопка и др. Написанные программы *управляются событиями*, в частности событиями мыши и клавиатуры. Поскольку большинство существующих программ взаимодействует с дисковой памятью, в книге приведены примеры чтения и записи файлов в долговременную память. Описаны решения *самых типичных задач*, которые встречаются в практике программирования, в частности работа *с графикой* и *буфером обмена*. Рассмотрены манипуляции табличными данными, использование обозревателя Web-страниц для отображения различных данных. Приведены примеры программирования с применением функций (методов) объектных библиотек систем Microsoft Office, MATLAB и AutoCAD. Разобраны вопросы обработки баз данных SQL Server и MS Access с помощью технологии ADO.NET. Рассмотрены методы обработки различных источников данных с использованием технологии LINQ. Приведены примеры программирования *Web-ориентированных приложений*, а также использования и разработки *Web-сервисов*. Новейшая технология WPF представлена несколькими выразительными примерами.

Несколько слов об особенностях книги. Спросите у любого программиста, *как он работает* (творит...) над очередной поставленной ему задачей. Он вам скажет, что всю задачу он мысленно *разбивает на фрагменты* и вспоминает, в каких уже решенных им задачах он *уже сталкивался с подобной ситуацией*. Далее он просто копирует фрагменты отлаженного программного кода и *вставляет их в новую задачу*. Сборник таких фрагментов (более 120 примеров) содержит данная книга. Автор пытался выделить *наиболее типичные*, актуальные задачи и решить их, с одной стороны, максимально эффективно, а с другой стороны, *кратко* и *выразительно*. Вместе с книгой читателю предлагается компакт-диск с рассмотренными в книге примерами.

Самая серьезная проблема в программировании больших, сложных программ — это *сложность, запутанность текстов*. Из-за запутанности программ имеются ошибки, нестыковки и проч. Как следствие — страдает производитель-

ность процесса создания программ *и их сопровождение*. Решение этой проблемы состоит в *структуризации* программ. Появление объектно-ориентированного программирования связано в большой степени со структуризацией программирования. Мероприятия для обеспечения большей структуризации — это проектирование программы как *иерархической структуры*, отдельные процедуры, входящие в программу, не должны быть *слишком длинными*, неиспользование операторов перехода `goto` и проч. Кроме того, современные системы программирования разрешают в названиях переменных, методов, свойств, событий, классов, объектов *использовать русские буквы*. Между тем современные программисты, как правило, *не используют* данную возможность, хотя когда вдруг в среде англоязычного текста появляются русские слова, это *вносит большую выразительность* в текст программы, и тем самым большую структуризацию. Программный код начинает от этого лучше читаться, *восприниматься человеком* (транслятору, компилятору — все равно).

Данная книга предназначена для начинающих программистов, программистов среднего уровня, а также для программистов, имеющих навыки разработки *на других языках* и желающих ускоренными темпами освоить новый для себя язык *Visual C# 2010*. Как пользоваться книгой? Эффективно пользоваться книгой можно, последовательно решая примеры в порядке их представления в книге, поскольку примеры расположены *от простого к более сложному*. И тем самым постепенно совершенствуя свои навыки программирования на *Visual C#*. А для программистов среднего уровня можно посоветовать искать *выборочно* именно те задачи, которые возникли у них при программировании их текущих задач.

Надеюсь, что читатель получит одновременно *удовольствие* и *пользу* от использования данной книги в своей работе и творчестве. Свои впечатления о данной книге присылайте по адресу **ziborov@ukr.net**, я с удовольствием их почитаю.

Глава 1

Простейшие программы с экранной формой и элементами управления

Пример 1. Форма, кнопка, метка и диалоговое окно

После инсталляции системы программирования Visual Studio 2010, включающей в себя Visual C# 2010, загрузочный модуль системы devenv.exe будет, скорее всего, расположен в папке: C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE.

Целесообразно создать ярлык на рабочем столе для запуска Visual C#. После запуска увидим начальный пользовательский интерфейс, показанный на рис. 1.1.

Рис. 1.1. Фрагмент стартовой страницы системы Visual Studio 2010

Чтобы запрограммировать какую-либо задачу, необходимо в пункте меню **File** выполнить команду **New Project**. В появившемся окне **New Project** в левой колонке находится список инсталлированных шаблонов (**Installed Templates**). Среди

них — шаблоны языков программирования, встроенных в Visual Studio, в том числе Visual Basic, Visual C#, Visual C++, Visual F# и др. Нам нужен язык Visual C#. В средней колонке выберем шаблон (Templates) **Windows Forms Application C#** и щелкнем на кнопке **OK**. В результате увидим окно, представленное на рис. 1.2.

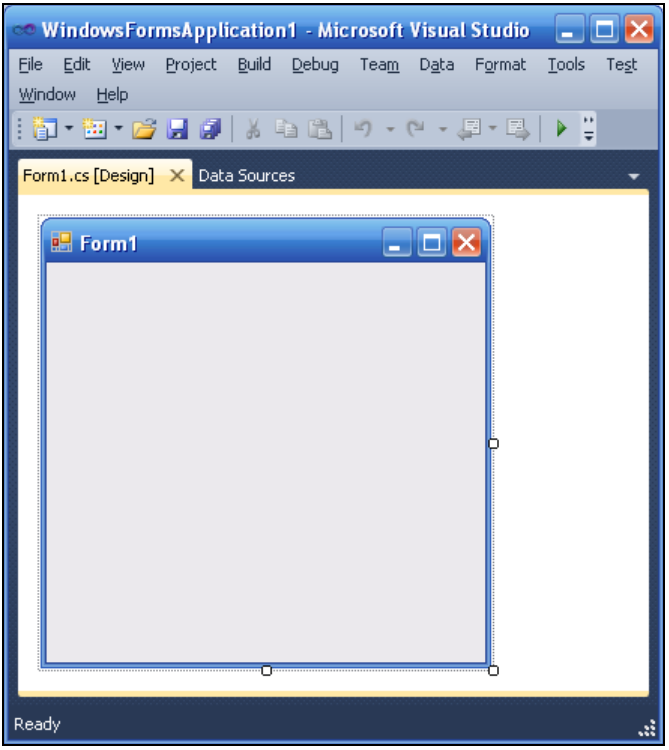

Рис. 1.2. Окно для проектирования пользовательского интерфейса

В этом окне изображена *экранная форма* — **Form1**, в которой программисты располагают различные компоненты графического интерфейса пользователя или, как их иначе называют, элементы управления. Это поля для ввода текста **TextBox**, командные кнопки **Button**, строчки текста в форме — метки **Label**, которые не могут быть отредактированы пользователем, и прочие элементы управления. Причем здесь используется самое современное так называемое *визуальное программирование*, предполагающее простое перетаскивание мышью из панели элементов **Toolbox**, где расположены всевозможные элементы управления, в форму. Таким образом, стараются свести к минимуму непосредственное написание программного кода.

Ваша первая программа будет отображать такую экранную форму, в которой будет что-либо написано, например "Microsoft Visual C# 2010", также в форме будет расположена командная кнопка с надписью "Нажми меня". При нажатии кнопки появится диалоговое окно с сообщением "Всем привет!"

Написать такую программку — вопрос 2—3 минут. Но вначале я хотел бы буквально двумя словами объяснить современный объектно-ориентированный подход к программированию. Подход заключается в том, что в программе все, что может быть названо именем существительным, называют *объектом*. Так, в нашей программе мы имеем четыре объекта: форму **Form**, надпись на форме **Label**, кнопку **Button** и диалоговое окно `MessageBox` с текстом "Всем привет!" (окно с приветом). Итак, добавьте метку и кнопку на форму примерно так, как показано на рис. 1.3.

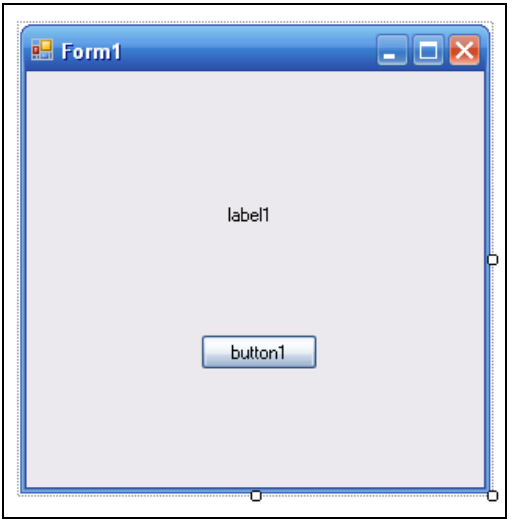

Рис. 1.3. Форма первого проекта

Любой такой объект можно создавать самому, а можно пользоваться готовыми объектами. В данной задаче мы пользуемся готовыми визуальными объектами, которые можно перетаскивать мышью из панели элементов управления **Toolbox**. В этой задаче нам нужно знать, что каждый объект имеет свойства (properties). Например, свойствами кнопки являются (рис. 1.4): имя кнопки (`Name`) — `button1`, надпись на кнопке (`Text`), расположение кнопки (`Location`) в системе координат формы `X`, `Y`, размер кнопки `Size` и т. д. Свойств много, их можно увидеть, если щелкнуть правой кнопкой мыши в пределах формы и выбрать в контекстном меню команду **Properties**; при этом появится панель свойств **Properties** (см. рис. 1.4).

Указывая мышью на все другие элементы управления в форме, можно просмотреть их свойства: формы `Form1` и надписи в форме — метки `label1`.

Вернемся к нашей задаче. Для объекта `label1` выберем свойство `Text` и напишем напротив этого поля "Microsoft Visual C# 2010" (вместо текста `label1`). Для объекта `button1` также в свойстве `Text` напишем "Нажми меня".

Кроме того, что объекты имеют свойства, следует знать, что объекты обрабатываются событиями. Событием, например, является щелчок на кнопке, щелчок в пределах формы, загрузка (`Load`) формы в оперативную память при старте программы и проч.

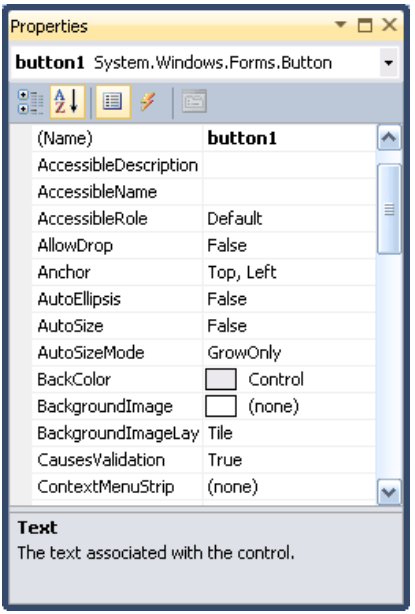

Рис. 1.4. Свойства кнопки `button1`

Управляют тем или иным событием посредством написания процедуры обработки события в программном коде. Для этого вначале нужно получить *"пустой" обработчик события*. В нашей задаче единственным событием, которым мы управляем, является щелчок на командной кнопке. Для получения пустого обработчика этого события следует в свойствах кнопки `button1` (см. рис. 1.4) щелкнуть на значке молнии **Events** (События) и в списке всех возможных событий кнопки `button1` выбрать двойным щелчком событие **Click**. После этого мы попадем на вкладку программного кода **Form1.cs** (рис. 1.5).

При этом управляющая среда Visual C# 2010 сгенерировала тот самый пустой обработчик события `button1_Click`:

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

в фигурных скобках которого мы можем написать команды, подлежащие выполнению после щелчка на кнопке. Вы видите, что у нас теперь две вкладки: **Form1.cs** и **Form1.cs [Design]**, т. е. вкладка программного кода и вкладка визуального проекта программы (другое название этой вкладки — *дизайнер формы*). Переключаться между ними можно мышью или нажатием комбинации клавиш <Ctrl>+<Tab>, как это принято обычно между вкладками в Windows, а также функциональной клавишей <F7>.

Напомню, что после щелчка на кнопке должно появиться диалоговое окно, в котором написано: "Всем привет!" Поэтому в фигурных скобках обработчика события напишем:

```
MessageBox.Show("Всем привет!");
```

Здесь вызывается метод (программа) `Show` объекта `MessageBox` с текстом "Всем привет!" Таким образом, я здесь "нечаянно" проговорился о том, что объекты кроме свойств имеют также и *методы*, т. е. программы, которые обрабатывают объек-

ты. Кстати, после каждого оператора в С-программах[1] ставят *точку с запятой*. Это вместе с фигурными скобками по форме отличает С-программу от программных кодов на других языках программирования.

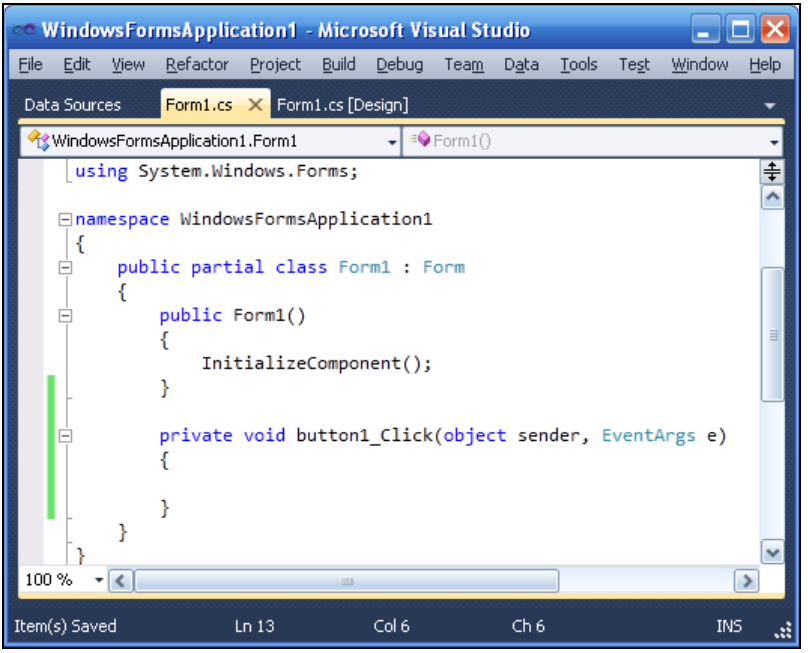

Рис 1.5. Вкладка программного кода

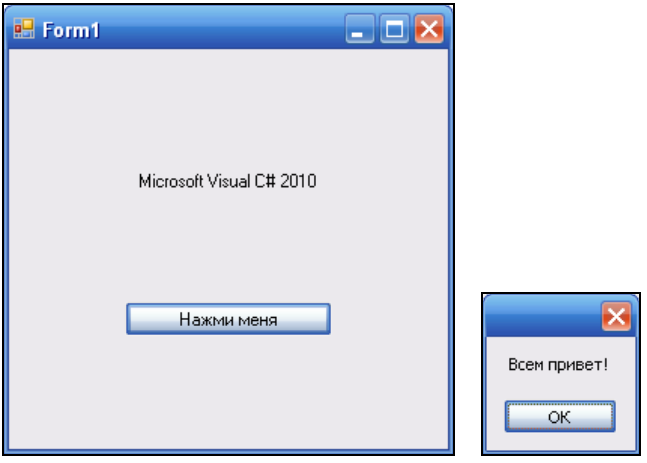

Рис. 1.6. Работающая программа

[1] Автор намеренно здесь указал "С", а не "С#", чтобы подчеркнуть особенность синтаксиса *любой* Си-программы, будь то С++, или С#, или Turbo-C.

Таким образом, мы написали *процедуру обработки события* щелчка (`Click`) на кнопке `button1`. Теперь нажмем клавишу <F5> и проверим работоспособность программы (рис. 1.6).

Поздравляю, вы написали свою первую программу на Visual C#!

Пример 2. Событие *MouseHover*

Немного усложним предыдущую задачу. Добавим еще одну обработку события `MouseHover` мыши для объекта `label1`. Событие `MouseHover` наступает тогда, когда пользователь указателем мыши "зависает" над каким-либо объектом, причем именно "зависает", а не просто перемещает мышь над объектом (от англ. *hover* — реять, парить). Есть еще событие `MouseEnter` (Войти), когда указатель мыши *входит в пределы* области элемента управления (в данном случае метки `label1`).

Чтобы добавить обработку события `MouseHover` мыши для объекта `label1`, следует так же, как это мы делали при обработке события "щелчок на кнопке", в панели **Properties** щелкнуть на значке молнии (**Events**) и двойным щелчком выбрать для объекта `label1` событие `MouseHover`. При этом осуществится переход на вкладку программного кода **Form1.cs** и среда Visual Studio 2010 сгенерирует простой обработчик события:

```
private void label1_MouseHover(object sender, EventArgs e) {  }
```

Между фигурными скобками вставим вызов диалогового окна:

```
MessageBox.Show("Событие Hover!");
```

Теперь проверим возможности программы: нажимаем клавишу <F5>, "зависаем" указателем мыши над `label1`, щелкаем на кнопке `button1`. Все работает!

А теперь я буду немного противоречить сам себе. Я говорил про визуальную технику программирования, направленную на минимизацию написания программного кода. А сейчас хочу сказать про *наглядность*, *оперативность*, *технологичность* работы программиста. Посмотрите на свойства каждого объекта в панели **Properties**. Вы видите, как много строчек. Если вы меняете какое-либо свойство, то оно будет выделено жирным шрифтом. Удобно! Но все-таки еще более удобно свойства объектов *назначать* (устанавливать) *в программном коде*. Почему?

Каждый программист имеет в своем арсенале множество уже отлаженных фрагментов, которые он использует в своей очередной новой программе. Программисту стоит лишь вспомнить, где он программировал ту или иную ситуацию. Программа, которую написал программист, имеет свойство быстро забываться. Если вы посмотрите на строчки кода, которые писали три месяца назад, то будете ощущать, что многое забыли; если прошел год, то вы смотрите на написанную вами программу, как на чужую. Поэтому при написании программ на первое место выходят *понятность*, *ясность*, *очевидность* написанного программного кода. Для этого каждая система программирования имеет какие-либо средства. Кроме того, сам программист придерживается некоторых правил, помогающих ему работать *производительно* и *эффективно*.

Назначать свойства объектов в программном коде удобно или сразу после инициализации компонентов формы (после процедуры `InitializeComponent`), или при обработке события `Form1_Load`, т. е. события загрузки формы в оперативную память при старте программы. Получим простой обработчик этого события. Для этого, как и в предыдущих случаях, можно выбрать нужное событие в панели свойств объекта, а можно еще проще: дважды щелкнуть в пределах проектируемой формы на вкладке **Form1.cs [Design]**. В любом случае получаем пустой обработчик события на вкладке программного кода. Заметим, что для формы таким умалчиваемым событием, для которого можно получить пустой обработчик двойным щелчком, является событие загрузки формы `Form1_Load`, для командной кнопки **Button** и метки **Label** таким событием является одиночный щелчок мышью на этих элементах управления. То есть если дважды щелкнуть в дизайнере формы по кнопке, то получим пустой обработчик `button1_Click` в программном коде, аналогично — для метки **Label**.

Итак, вернемся к событию загрузки формы, для него управляющая среда сгенерировала пустой обработчик:

```
private void Form1_Load(object sender, EventArgs e) { }
```

Здесь в фигурных скобках обычно вставляют свойства различных объектов и даже часто пишут много строчек программного кода. Здесь мы назначим свойству `Text` объекта `label1` значение "Microsoft Visual C# 2010":

```
label1.Text = "Microsoft Visual C# 2010";
```

Аналогично для объекта `button1`:

```
button1.Text = "Нажми меня!";
```

Совершенно необязательно писать каждую букву приведенных команд. Например, для первой строчки достаточно написать "la", уже это вызовет появление раскрывающегося меню, где вы сможете выбрать нужные для данного контекста ключевые слова. Это очень мощное и полезное современное средство, называемое IntellSense, для редактирования программного кода! Если вы от Visual Studio 2010 перешли в другую систему программирования, в которой отсутствует подобный сервис, то будете ощущать сильный дискомфорт.

Вы написали название объекта `label1`, поставили точку. Теперь вы видите раскрывающееся меню, где можете выбрать либо нужное свойство объекта, либо метод (т. е. подпрограмму). В данном случае выберите свойство `Text`.

Как видите, не следует пугаться слишком длинных ключевых слов, длинных названий объектов, свойств, методов, имен переменных. Система подсказок современных систем программирования значительно облегчает всю нетворческую работу. Вот почему в современных программах можно наблюдать такие длинные имена ключевых слов, имен переменных и проч. Я призываю вас, уважаемые читатели, также использовать в своих программах для названия переменных, объектов наиболее ясные, полные имена, причем можно на вашем родном русском языке. Потому что на первое место выходят ясность, прозрачность программирования, а громоздкость названий с лихвой компенсируется системой подсказок.

Далее хотелось бы, чтобы слева вверху формы на синем фоне (в так называемой строке заголовка) была не надпись "Form1", а что-либо осмысленное. Например, слово "Приветствие". Для этого ниже присваиваем эту строку свойству `Text` формы. Поскольку мы изменяем свойство объекта `Form1` внутри подпрограммы обработки события, связанного с формой, следует к форме обращаться через ссылку `this`: `this.Text = "Приветствие"` или `base.Text = "Приветствие"`.

После написания последней строчки кода мы должны увидеть на экране программный код, представленный в листинге 1.1.

Листинг 1.1. Программирование событий

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }
   // Обработка события загрузки формы:
      private void Form1_Load(object sender, EventArgs e)
      {
         this.Text = "Приветствие";
         label1.Text = "Microsoft Visual C# 2010";
         button1.Text = "Нажми меня";
      }
   // Обработка события щелчок на кнопке:
      private void button1_Click(object sender, EventArgs e)
      {
         MessageBox.Show("Всем привет!");
      }
   // Обработка события, когда указатель мыши "завис" над меткой:
      private void label1_MouseHover(object sender, EventArgs e)
```

```
        {
            MessageBox.Show("Событие Hover!");
        }
    }
}
```

Комментарии, поясняющие работу программы, в окне редактора кода будут выделены зеленым цветом, чтобы в тексте выразительно отделять его от прочих элементов программы. В С-подобных языках комментарий пишут после двух слэшей (`//`) или внутри пар `/* */`. Уважаемые читатели, даже если вам кажется весьма очевидным то, что вы пишете в программном коде, *напишите комментарий*. Как показывает опыт, даже весьма очевидный замысел программиста забывается удивительно быстро. Человеческая память отметает все, что по оценкам организма считается ненужным. Кроме того, даже если текст программы вполне ясен, в начале программы должны быть описаны ее назначение и способ использования, т. е. как бы "преамбула" программы. Далее в примерах мы будем следовать этому правилу.

На рис. 1.7 приведен фрагмент работы программы.

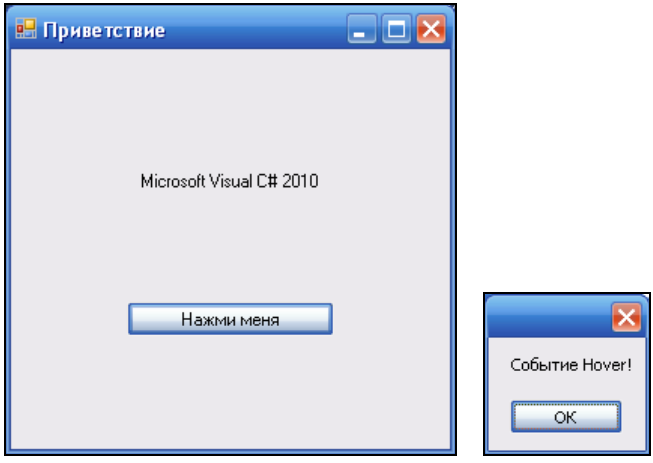

Рис. 1.7. Работа программы

Обычно в редакторах программного кода используется моноширинный шрифт, поскольку все символы такого шрифта имеют одинаковую ширину, в том числе точка и прописная русская буква "Ш". По умолчанию в редакторе программного кода C# 2010 задан шрифт Consolas. Однако если пользователь привык к шрифту Courier New, то его настройку можно изменить, выбрав меню **Tools | Options | Environment | Fonts and Colors**.

Теперь закроем проект (**File | Close Project**). Система предложит нам сохранить проект, сохраним проект под именем Hover. Теперь программный код этой программы можно посмотреть, открыв решение Hover.sln в папке Hover.

Пример 3. Ввод данных через текстовое поле *TextBox* с проверкой типа методом *TryParse*

При работе с формой очень часто ввод данных организуют через элемент управления "текстовое поле" **TextBox**. Напишем типичную программу, которая вводит через текстовое поле число, при нажатии командной кнопки извлекает из него квадратный корень и выводит результат на метку **Label**. В случае ввода не числа сообщает пользователю об этом.

Решая сформулированную задачу, запускаем Visual Studio, выбираем пункт меню **File | New | Project**, затем — шаблон **Windows Forms Application Visual C#** и щелкаем на кнопке **OK.** Далее из панели элементов управления **Toolbox** (если в данный момент вы не видите панель элементов, то ее можно добавить, например, с помощью комбинации клавиш <Ctrl>+<Alt>+<X> или меню **View | Toolbox**) в форму указателем мыши перетаскиваем текстовое поле **TextBox**, метку **Label** и командную кнопку **Button**. Таким образом, в форме будут находиться три элемента управления.

Теперь перейдем на вкладку программного кода, для этого правой кнопкой мыши вызовем контекстное меню и выберем в нем пункт **View Code**. Здесь сразу после инициализации компонентов формы, т. е. после вызова процедуры `InitializeComponent` задаем свойствам формы (к форме обращаемся посредством ссылки `this` или `base`), кнопкам `button1` и текстового поля `textBox1`, метке `label1` следующие значения:

```
this.Text = "Извлечение квадратного корня";
button1.Text = "Извлечь корень";
textBox1.Clear();     // Очистка текстового поля
label1.Text = null;  // или = string.Empty;
```

Нажмем клавишу <F5> для выявления возможных опечаток, т. е. синтаксических ошибок и предварительного просмотра дизайна будущей программы.

Далее программируем событие `button1_Click` — щелчок мышью на кнопке **Извлечь корень**. Создать пустой обработчик этого события удобно, дважды щелкнув мышью на этой кнопке. Между двумя появившимися строчками программируем диагностику правильности вводимых данных, конвертирование строковой переменной в переменную типа `Single` и непосредственное извлечение корня (листинг 1.2).

Листинг 1.2. Извлечение корня с проверкой типа методом `TryParse`

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
```

```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

// Программа вводит через текстовое поле число, при щелчке на командной кнопке
// извлекает из него квадратный корень и выводит результат на метку label1.
// В случае ввода не числа сообщает пользователю об этом,
// выводя красным цветом предупреждение также на метку label1.

namespace Корень
{
   public partial class Form1 : Form
   {
      public Form1()
      {  // Инициализация компонентов формы
         InitializeComponent();
         button1.Text = "Извлечь корень";
         label1.Text = null;    // или = string.Empty;
         this.Text = "Извлечение квадратного корня";
         textBox1.Clear();      // очистка текстового поля
         textBox1.TabIndex = 0; // установка фокуса в текстовом поле
      }

      private void button1_Click(object sender, EventArgs e)
      {  // Обработка щелчка на кнопке "Извлечь корень"
         Single X;   // - из этого числа будем извлекать корень
         // Преобразование из строковой переменной в Single:
         bool Число_ли = Single.TryParse(textBox1.Text,
            System.Globalization.NumberStyles.Number,
            System.Globalization.NumberFormatInfo.CurrentInfo, out X);
         // Второй параметр - это разрешенный стиль числа (Integer,
         // шестнадцатиричное число, экспоненциальный вид числа и проч.).
         // Третий параметр форматирует значения на основе текущего
         // языка и региональных параметров из
         // Панели управления -> Язык и региональные стандарты
         // число допустимого формата;
         // метод возвращает значение в переменную X
         if (Число_ли == false)
         {  // Если пользователь ввел не число:
            label1.Text = "Следует вводить числа";
            label1.ForeColor = Color.Red;  // красный цвет текста на метке
```

```
            return;                          // выход из процедуры
        }
        Single Y = (Single)Math.Sqrt(X);  // извлечение корня
        label1.ForeColor = Color.Black;   // черный цвет текста на метке
        label1.Text = string.Format("Корень из {0} равен {1:F5}", X, Y);
    }
  }
}
```

Здесь при обработке события "щелчок мышью" на кнопке **Извлечь корень** проводится проверка, введено ли число в текстовом поле. Проверка осуществляется с помощью функции `TryParse`. Первым параметром метода `TryParse` является анализируемое поле `textBox1.Text`. Второй параметр — это разрешаемый для преобразования стиль числа, он может быть целого типа (`Integer`), шестнадцатеричным (`HexNumber`), представленным в экспоненциальном виде и проч. Третий параметр указывает, на какой основе формируется допустимый формат, в нашем случае мы использовали `CurrentInfo`, т. е. на основе текущего языка и региональных параметров. По умолчанию при инсталляции руссифицированной версии Windows разделителем целой и дробной части числа является запятая. Однако эту установку можно изменить, если в Панели управления выбрать значок **Язык и региональные стандарты**, затем на вкладке **Региональные параметры** щелкнуть на кнопке **Настройка** и на появившейся новой вкладке указать в качестве разделителя целой и дробной частей точку вместо запятой. В обоих случаях (и для запятой, и для точки) метод `TryParse` будет работать так, как указано на вкладке **Региональные параметры**.

Четвертый параметр метода `TryParse` возвращает результат преобразования. Кроме того, функция `TryParse` возвращает булеву переменную `true` или `false`, которая сообщает, успешно ли выполнено преобразование. Как видно из текста программы, если пользователь ввел не число (например, введены буквы), то на метку `label1` выводится красным цветом текст "Следует вводить числа". Далее, поскольку ввод неправильный, организован выход из программы обработки события `button1_Click` с помощью оператора `return`. На рис. 1.8 показан фрагмент работы программы.

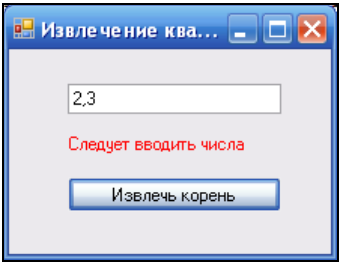

Рис 1.8. Фрагмент работы программы

Как видно из рисунка, функция `TryParse` не восприняла введенные символы "2,3" как число, поскольку автор специально для демонстрации данного примера указал на вкладке **Региональные параметры** точку в качестве разделителя целой и дробной частей.

Если пользователь ввел все-таки число, то будет выполняться следующий оператор извлечения квадратного корня `Math.Sqrt(X)`. Математические функции Visual Studio 2010 являются методами класса `Math`. Их можно увидеть, набрав `Math` и поставив точку (`.`). В выпадающем списке вы увидите множество математических функций: `Abs`, `Sin`, `Cos`, `Min` и т. д. и два свойства — две константы `E = 2.71...` (основание натуральных логарифмов) и `PI = 3,14...` (число диаметров, уложенных вдоль окружности). Функция `Math.Sqrt(X)` возвращает значение типа `double` (двойной точности с плавающей запятой), которое *приводим* с помощью неявного преобразования (`Single`) к переменной одинарной точности.

Последней строчкой обработки события `button1_Click` является формирование строки `label1.Text` с использованием метода `string.Format`. Использованный формат `"Корень из {0} равен {1:F5}"` означает: взять нулевой выводимый элемент, т. е. переменную `X`, и записать эту переменную вместо фигурных скобок; после чего взять первый выводимый элемент, т. е. `Y`, и записать его вместо вторых фигурных скобок в формате с фиксированной точкой и пятью десятичными знаками после запятой.

Нажав клавишу <F5>, проверяем, как работает программа.

Результат работающей программы представлен на рис. 1.9.

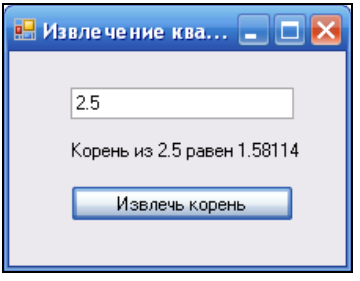

Рис. 1.9. Извлечение квадратного корня

Если появились ошибки, то работу программы следует проверить отладчиком — клавиши <F8> или <F11>. В этом случае управление останавливается на каждом операторе, и вы можете проверить значение каждой переменной, наводя указатель мыши на переменные. Можно выполнить программу до определенной программистом точки *(точки останова)*, используя, например, клавишу <F9> или оператор `Stop`, и в этой точке проверить значения необходимых переменных.

Убедиться в работоспособности этой программы можно, открыв соответствующее решение в папке Корень.

Пример 4. Ввод пароля в текстовое поле и изменение шрифта

Это очень маленькая программа для ввода пароля в текстовое поле, причем при вводе вместо вводимых символов некто, "находящийся за спиной пользователя", увидит только звездочки. Программа состоит из формы, текстового поля **TextBox**, метки **Label**, куда для демонстрации возможностей мы будем копировать пароль (паспорт, т. е. секретные слова) и командной кнопки **Button — Покажи паспорт**.

Перемещаем в форму все названные элементы управления. Текст программы приведен в листинге 1.3.

Листинг 1.3. Ввод пароля

```
// Программа для ввода пароля в текстовое поле, причем при вводе вместо
// вводимых символов некто, "находящийся за спиной пользователя",
// увидит только звездочки

using System;
using System.Drawing;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются
// в данной программе

namespace Passport
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         base.Text = "Введи пароль";
         textBox1.Text = null; textBox1.TabIndex = 0;
         textBox1.PasswordChar = '*';
         textBox1.Font = new Font("Courier New", 9.0F);
         // или textBox1.Font = new Font(FontFamily.GenericMonospace, 9.0F);
         label1.Text = string.Empty;
         label1.Font = new Font("Courier New", 9.0F);
         button1.Text = "Покажи паспорт";
      }
      private void button1_Click(object sender, EventArgs e)
      {  // Обработка события "щелчок на кнопке"
```

```
            label1.Text = textBox1.Text;
        }
    }
}
```

Как видно из текста программы, сразу после инициализации компонентов формы, т. е. после вызова процедуры `InitializeComponent`, очищаем текстовое поле и делаем его "защищенным от посторонних глаз" с помощью свойства `textBox1.PasswordChar`, каждый введенный пользователем символ маскируется символом звездочки (`*`). Далее мы хотели бы для большей выразительности и читабельности программы, чтобы вводимые звездочки и результирующий текст имели одинаковую длину. Все символы шрифта Courier New имеют одинаковую ширину, поэтому его называют моноширинным шрифтом. Кстати, используя именно этот шрифт, удобно программировать таблицу благодаря одинаковой ширине букв этого шрифта. Еще одним широко используемым моноширинным шрифтом является шрифт Consola. Задаем шрифт, используя свойство `Font` обоих объектов: `textBox1` и `label1`. Число `9.0` означает размер шрифта. Свойство текстового поля `TabIndex = 0` обеспечивает передачу фокуса при старте программы именно в текстовое поле.

Осталось обработать событие `button1_Click` — щелчок на кнопке. Здесь — банальное присваивание текста из поля тексту метки. Программа написана, нажимаем клавишу <F5>. На рис. 1.10 приведен вариант работы данной программы.

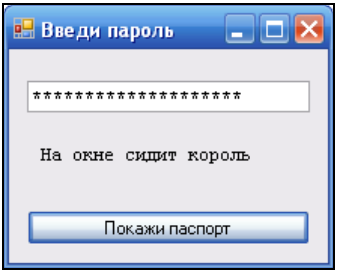

Рис 1.10. Вариант работы программы

При необходимости используйте отладчик (клавиша <F11> или <F10>) для *пошагового выполнения программы* и выяснения всех промежуточных значений переменных путем "зависания" указателя мыши над переменными.

Убедиться в работоспособности программы можно, открыв решение Passport.sln в папке Passport.

Пример 5. Управление стилем шрифта с помощью элемента управления *CheckBox*

Кнопка **CheckBox** (Флажок) также находится на панели элементов управления **Toolbox**. Флажок может быть либо установлен (содержит "галочку"), либо сброшен

(пустой). Напишем программу, которая управляет стилем шрифта текста, выведенного на метку **Label**. Управлять стилем будем посредством флажка **CheckBox**.

Используя панель элементов **Toolbox**, в форму поместим метку `label1` и флажок `checkBox1`. В листинге 1.4 приведен текст программы управления этими объектами.

Листинг 1.4. Управление стилем шрифта

```
// Программа управляет стилем шрифта текста, выведенного на метку
// Label, посредством флажка CheckBox
using System;
using System.Drawing;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются
// в данной программе

namespace CheckBox
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         this.Text = "Флажок CheckBox";
         checkBox1.Text = "Полужирный"; checkBox1.Focus();
         label1.Text = "Выбери стиль шрифта";
         label1.TextAlign = ContentAlignment.MiddleCenter;
         label1.Font = new System.Drawing.Font("Courier New", 14.0F);
      }
      private void checkBox1_CheckedChanged(object sender, EventArgs e)
      { // Изменение состояния флажка на противоположное
         if (checkBox1.Checked == true) label1.Font =
                     new Font("Courier New", 14.0F, FontStyle.Bold);
         if (checkBox1.Checked == false) label1.Font =
                     new Font("Courier New", 14.0F, FontStyle.Regular);
      }
   }
}
```

Сразу после вызова процедуры `InitializeComponent` задаем начальные значения некоторых свойств объектов `Form1` (посредством ссылки `this`), `label1` и `checkBox1`. Так, тексту флажка, выводимого с правой стороны, присваиваем значение "Полужирный". Кроме того, при старте программы фокус должен находиться

на флажке (`checkBox1.Focus();`), в этом случае пользователь может изменять установку флажка даже клавишей <Пробел>.

Текст метки — "Выбери стиль шрифта", выравнивание метки `TextAlign` задаем посередине и по центру (`MiddleCenter`) относительно всего того места, что предназначено для метки. Задаем шрифт метки Courier New (в этом шрифте все буквы имеют одинаковую ширину) размером 14 пунктов.

Изменение состояния флажка соответствует событию `CheckedChanged`. Чтобы получить пустой обработчик события `CheckedChanged`, следует дважды щелкнуть на элементе `checkBox1` вкладки **Form1.cs [Design]**. Между соответствующими строчками следует записать (см. текст программы): если флажок установлен (т. е. содержит "галочку") `Checked = true`, то для метки `label1` устанавливается тот же шрифт Courier New, 14 пунктов, но `Bold`, т. е. полужирный.

Далее — следующая строчка кода: если флажок не установлен, т. е. `checkBox1.Checked = false`, то шрифт устанавливается `Regular`, т. е. обычный. Очень часто эту ситуацию программируют, используя ключевое слово `else` (иначе), однако это выражение будет выглядеть более выразительно и понятно так, как написали мы.

Программа написана, нажмите клавишу <F5>. Проверьте работоспособность программы. В рабочем состоянии она должна работать примерно так, как показано на рис. 1.11.

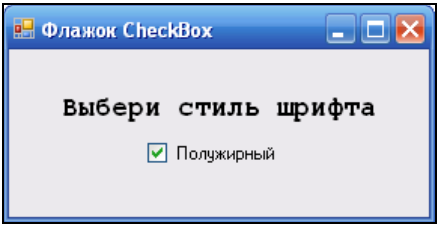

Рис. 1.11. Фрагмент работы программы управления стилем шрифта

Убедиться в работоспособности программы можно, открыв решение CheckBox1.sln в папке CheckBox1.

Пример 6. Побитовый оператор "исключающее ИЛИ"

Несколько изменим предыдущую программу в части обработки события `CheckedChanged` (Изменение состояния флажка). Вместо двух условий `if()` напишем один оператор:

```
label1.Font = new Font("Courier New", 14.0F,
                       label1.Font.Style ^ FontStyle.Bold);
```

Здесь каждый раз при изменении состояния флажка значение параметра `label1.Font.Style` сравнивается с одним и тем же значением `FontStyle.Bold`. Поскольку между ними стоит побитовый оператор `^` (исключающее ИЛИ), он будет назначать `Bold`, если текущее состояние `label1.Font.Style` "не Bold". А если `label1.Font.Style` пребывает в состоянии "Bold", то оператор `^` будет назначать состояние "не Bold". Этот оператор еще называют логическим `XOR`.

Таблица истинности логического `XOR` такова:

```
A Xor B = C
0 Xor 0 = 0
1 Xor 0 = 1
0 Xor 1 = 1
1 Xor 1 = 0
```

В нашем случае мы имеем всегда `B = 1` (`FontStyle.Bold`), а `A` (`Label1.Font.Style`) попеременно то `Bold`, то `Regular` (т. е. "не Bold"). Таким образом, оператор `^` всегда будет назначать противоположное тому, что записано в `label1.Font.Style`.

Как видно, применение побитового оператора привело к существенному уменьшению количества программного кода. Использование побитовых операторов может значительно упростить написание программ со сложной логикой.

Посмотрите, как работает программа, нажав клавишу <F5>.

Теперь добавим в форму еще один элемент управления **CheckBox**. Мы собираемся управлять стилем шрифта `FontStyle` двумя флажками. Один, как и прежде, задает полужирный стиль `Bold` или обычный `Regular`, а второй задает наклонный `Italic` или возвращает в `Regular`. Текст новой программы приведен в листинге 1.5.

Листинг 1.5. Усовершенствованный программный код

```
// Побитовый оператор "^" - исключающее ИЛИ
using System;
using System.Drawing;
using System.Windows.Forms;
// Другие директивы using удалены, поскольку они не используются
// в данной программе

namespace CheckBox2
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
         this.Text = "Флажок CheckBox";
         checkBox1.Text = "Полужирный"; checkBox2.Text = "Наклонный";
```

```
            label1.Text = "Выбери стиль шрифта";
            label1.TextAlign = ContentAlignment.MiddleCenter;
            label1.Font = new Font("Courier New", 14.0F);
        }
        private void checkBox1_CheckedChanged(object sender, EventArgs e)
        {
            label1.Font = new System.Drawing.Font(
                "Courier New", 14.0F, label1.Font.Style ^ FontStyle.Bold);
        }
        private void checkBox2_CheckedChanged(object sender, EventArgs e)
        {
            label1.Font = new System.Drawing.Font(
                "Courier New", 14.0F, label1.Font.Style ^ FontStyle.Italic);
        }
    }
}
```

Как видно, здесь принципиально ничего нового нет, только лишь добавлена обработка события изменения состояния флажка `CheckedChanged` для `CheckBox2`. Фрагмент работы программы можно увидеть на рис. 1.12.

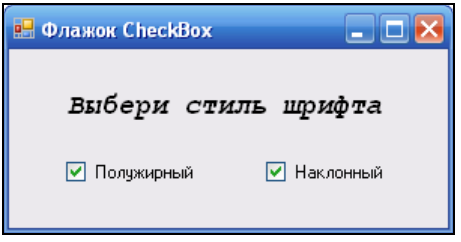

Рис. 1.12. Фрагмент работы усовершенствованной программы

Убедиться в работоспособности программы можно, открыв решение CheckBox2.sln в папке CheckBox2.

Пример 7. Вкладки *TabControl* и переключатели *RadioButton*

Вкладки используются для организации управления и оптимального расположения экранного пространства. Выразительным примером использования вкладок является диалоговое окно **Свойства обозревателя** Internet Explorer. То есть если требуется отобразить большое количество управляемой информации, то весьма уместно использовать вкладки **TabControl**.

Поставим задачу написать программу, позволяющую выбрать текст из двух вариантов, задать цвет и размер шрифта этого текста на трех вкладках **TabControl** с использованием переключателей **RadioButton**.

Фрагмент работы программы приведен на рис. 1.13.

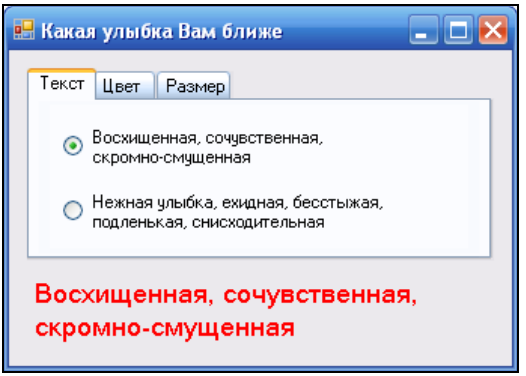

Рис. 1.13. Программа с переключателями и вкладками

Программируя поставленную задачу, создадим новый проект **Windows Forms Application C#**, назовем его, например, ВкладкиTabControl, получим стандартную форму. Затем, используя панель элементов **Toolbox**, в форму перетащим мышью элемент управления **TabControl**. Как видно, по умолчанию имеем две вкладки, а по условию задачи, как показано на рис. 1.13, три вкладки. Добавить третью вкладку можно в конструкторе формы, а можно программно.

Вначале покажем, как добавить третью вкладку в конструкторе. Для этого в свойствах (окно **Properties**) элемента управления `TabControl1` выбираем свойство `TabPages`, в результате попадаем в диалоговое окно **TabPage Collection Edit**, где добавляем (кнопка **Add**) третью вкладку (первые две присутствуют по умолчанию). Эти вкладки нумеруются от нуля, т. е. третья вкладка будет распознаваться как `TabPages(2)`. Название каждой вкладки будем указывать в программном коде.

Для того чтобы читатель в большей степени мог управлять всем процессом, мы покажем, как добавить третью вкладку не в конструкторе, а в программном коде сразу после вызова процедуры `InitializeComponent` (листинг 1.6). Однако, прежде чем перейти на вкладку программного кода, для каждой вкладки выбираем из панели **Toolbox** по два переключателя **RadioButton**, а в форму перетаскиваем метку **Label**. Теперь через щелчок правой кнопкой мыши в пределах формы переключаемся на редактирование программного кода.

Листинг 1.6. Использование вкладок и переключателей

```
// Программа, позволяющая выбрать текст из двух вариантов, задать цвет
// и размер шрифта для этого текста на трех вкладках TabControl
// с использованием переключателей RadioButton
```