ПАВЕЛ АГУРОВ

bhv®

C#

РАЗРАБОТКА КОМПОНЕНТОВ В MS VISUAL STUDIO 2005/2008

- РАЗРАБОТКА И ОТЛАДКА КОМПОНЕНТОВ
- КОМПОНЕНТЫ WIN FORMS
- ДИЗАЙНЕРЫ, КОНВЕРТЕРЫ ТИПА, СЕРВИСЫ, РЕДАКТОРЫ ТИПОВ
- ЛИЦЕНЗИРОВАНИЕ И РАСПРОСТРАНЕНИЕ
- ПРАКТИЧЕСКИЕ ПРИМЕРЫ

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

УДК 681.3.068+800.92
ББК 32.973.26-018.1
А27

Агуров П. В.

А27 С#. Разработка компонентов в MS Visual Studio 2005/2008. — СПб.: БХВ-Петербург, 2008. — 480 с.: ил. + CD-ROM — (Профессиональное программирование)

ISBN 978-5-9775-0295-5

Книга содержит всю необходимую информацию для создания полноценных компонентов Win Forms на языке C# в MS Visual Studio 2005/2008, начиная с разработки и отладки и заканчивая лицензированием и распространением. Рассмотрены дизайнеры, конвертеры типа, сервисы, редакторы типов и многое другое. Информация о каждом классе, описанном в книге, сопровождается примером его использования. Весь программмный код является авторской разработкой и проверен на практике. На компакт-диске приведены примеры из книги.

Для программистов

УДК 681.3.068+800.92
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	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

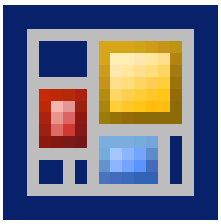
Лицензия ИД № 02429 от 24.07.00. Подписано в печать 23.05.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 38,7.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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

ISBN 978-5-9775-0295-5

Оглавление

ГЛАВА 1



Что такое компоненты

Компонент (от лат. *componens* — составляющий) — составная часть, элемент чего-либо.
(Выписка из словаря)

Коротко говоря, компонент — это "кирпичик" программы, состоящий из свойств (properties), методов (methods) и событий (events). *Свойства* дают возможность управлять видом и поведением компонента, *методы* — использовать возможности, предоставляемые компонентом, а *события* — реагировать на происходящие внутри компонента события, программировать реакцию компонента на внешние события и т. д.

Разработка программы с помощью компонентов называется *компонентно-ориентированной* разработкой. В этой главе я постараюсь описать, какие преимущества дает такой подход и какие подводные камни ждут нас на этом пути.

1.1. Компоненты: достоинства и недостатки

Компонентно-ориентированная разработка имеет свои сильные и слабые стороны. Несомненными достоинствами является повторная используемость кода, согласованность пользовательского интерфейса, возможность быстрой и продуктивной разработки программ. Именно компоненты позволяют программистам составлять конечный продукт из "кирпичиков", не вдаваясь в детали реализации конкретного компонента. Конечно, наборы классов, используемые при объектно-ориентированном подходе, тоже дают возможность повторного использования кода, но компоненты делают повторное использование кода совершенно естественным.

Если при разработке системы все программисты команды пользуются одним и тем же набором визуальных компонентов, то, само собой, интерфейс у программы будет выполнен в едином стиле. Более того, поменяв, например, вид одного из компонентов, мы изменим его вид везде, где он используется.

Компоненты дают возможность независимой разработки частей интерфейса. Изменения внутри компонента не затрагивают код модулей, в которых он используется. Разработка нескольких независимых классов дает примерно те же результаты, за исключением одной проблемы. Среди программистов средней квалификации наблюдается тенденция перемешивать функциональность классов, путая методы одного класса с другим. Компоненты разделяют код более качественно. Особо "одаренным" можно не отдавать исходный код компонентов, которые не касаются разрабатываемых ими модулей.

Свойства компонентов позволяют наиболее эффективно объяснить другому программисту, как использовать компонент. Специальные редакторы свойств позволяют быстро настроить вид и поведение компонентов.

Наконец, накопив достаточное количество компонентов, можно действительно быстро создать визуальный интерфейс программы, фактически, не написав более ни строчки кода!

Чем же придется заплатить за все это? Как это не удивительно звучит, но платить придется объектно-ориентированным подходом. Возможность гибкого управления поведением компонентов с помощью событий провоцирует написание "событийно-ориентированного" кода. Пусть, например, нам нужен компонент для отображения цветных строк. Объектно-ориентированный подход обязывает нас создать наследника класса `ListBox` и, перекрыв метод `Paint`, реализовать отрисовку цветных строк. Возможность реализовать событие `OnPaint` и не создавать никаких классов подталкивает многих программистов к использованию событий в ущерб объектно-ориентированному подходу. Я специально говорю "подталкивает", т. к. никто не мешает создать новый компонент, умеющий рисовать цветные строки на основе существующего компонента `ListBox`. Такой подход и будет наиболее верным — ведь такие компоненты можно использовать повторно!

Еще одна плата за удобство — необходимость иметь гибкие компоненты. Нет смысла писать компонент, рисующий только строки красного цвета. Такой компонент будет затруднительно использовать где-либо, кроме той программы, для которой он изначально предназначался. Гораздо правильнее написать компонент, рисующий строки заданного цвета, а сам цвет вынести в его свойства (или как-то хранить внутри самой строки). Вот такой компонент можно использовать в любой программе. Такая гибкость требует некоторых дополнительных усилий, необходимость затратить которые не всегда очевидна при разработке одной программы, но вполне окупается при повторном использовании компонента.

Подводя итог этому рассуждению, можно сказать, что цель разработки нового компонента — создание новой функциональности, являющейся независимой, но гибко настраиваемой частью, допускающей повторное использование.

1.2. Особенности разработки компонентов

В отличие от обычного класса, разработка компонента имеет особенности, которые следует учитывать, принимая решение о создании компонента.

Во-первых, пользователем компонента является такой же программист, как и мы с вами. Именно он, а не конечный пользователь программы, использует компонент. Конечному пользователю компонент достанется уже потом, когда программа будет готова.

Во-вторых, компонент имеет три интерфейса: интерфейс времени выполнения, интерфейс времени проектирования и программный интерфейс.

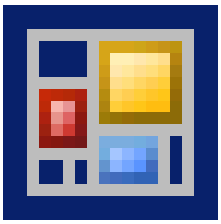
Интерфейс времени выполнения — это поведение компонента во время работы конечной программы. Например, компонент `MaskEdit` имеет поле ввода и умеет проверять правильность вводимых данных. Поведение времени выполнения складывается как из кода, заложенного в сам компонент, так и из кода, созданного пользователем компонента, с помощью обработчиков событий.

Интерфейс времени проектирования позволяет настроить вид и поведение компонента во время разработки. Стандартный дизайнер формы позволяет визуально настроить размер и положение компонента на форме, редактор свойств позволяет легко и быстро менять значения ключевых свойств, влияющих на вид и поведение компонента, а специальные редакторы самого компонента (если они есть) позволяют проводить тонкую настройку свойств компонента еще более быстро и эффективно. Конечно, программирование поведения компонента во время проектирования требует дополнительных усилий, иногда превышающих усилия на программирование самого компонента. Но, поверьте, хороший интерфейс времени проектирования того стоит!

Программный интерфейс — это свойства, методы и события, которые позволяют компонентам общаться между собой и с основной программой.

Основная особенность разработки компонента — сбалансированность гибкости компонента и времени, потраченного на его создание, плюс оценка затрат на разработку компонента с учетом возможности его повторного использования. Слишком гибкий компонент будет иметь очень много свойств, и его будет сложно настраивать. Наоборот, компонент, реализующий слишком частную ситуацию, будет бесполезен в плане повторного использования, и, возможно, затраты на его реализацию не будут иметь смысла.

ГЛАВА 2



Дизайнер MS Visual Studio

Пользователь всегда прав, но не должен об этом знать.

Нет, не пугайтесь, я не собираюсь детально разбирать здесь всю среду разработки MS Visual Studio, тратя бесценное книжное место. Но без некоторых деталей нам все же не обойтись. Как минимум, нам надо договориться об именовании основных окон среды. Итак, создадим новый проект типа **Windows Application** и посмотрим, что в нем есть полезного.

2.1. Среда Visual Studio

При создании нового приложения **Windows Application** среда открывает редактор формы и несколько окон (рис. 2.1), среди которых обычно (в зависимости от настроек среды) окна **Solution Explorer** (Обозреватель проекта), **Toolbox** (Инструменты) и **Properties** (Свойства). Именно эти три окна представляют для нас наибольший интерес.

В окне **Solution Explorer** (Обозреватель проекта) можно выбрать нужный для редактирования элемент: файл кода, форму и т. д. В соответствии с этим меняется редактор в левой части. Для формы отображается редактор формы, называемый еще *дизайнером формы*. Практически аналогичный редактор открывается при редактировании компонентов. При редактировании кода открывается текстовый редактор. Есть и другие редакторы (дизайнеры), но пока нам вполне достаточно знать про эти два.

В следующих разделах я расскажу про каждый из элементов среды более подробно.

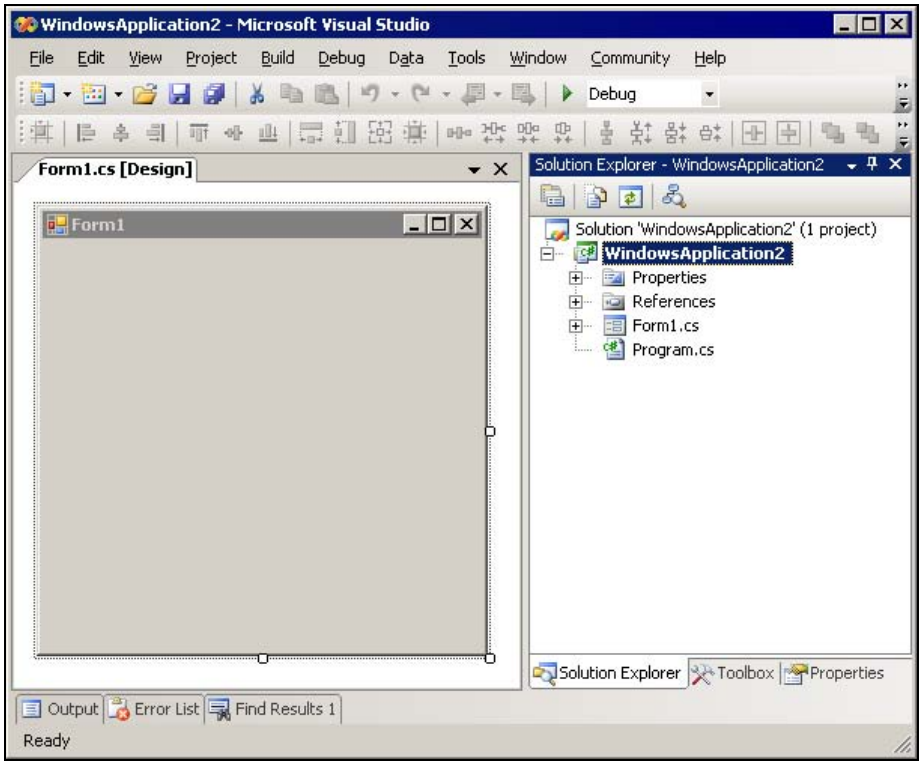


Рис. 2.1. Среда разработки Visual Studio

2.2. Дизайнер формы

В этом разделе я очень кратно опишу основные понятия, которые потребуются нам при работе с редактором формы и компонентов.

Дизайнер формы (form designer) имеет три маркера для изменения размера формы. С их помощью можно увеличить или уменьшить размер формы по высоте и ширине или одновременно по обоим параметрам, пропорционально (рис. 2.2).

Если на форме выбрать какой-нибудь элемент, то редактор показывает *маркеры* изменения размеров и положения этого элемента (glyph). Причем, как видно на рис. 2.3, маркеры меняются в зависимости от реальной возможности изменять размеры. Так, компонент `Button` можно менять по всем параметрам (во всех направлениях), а для компонента `CheckBox` доступно только изменение положения компонента на форме. Соответственно, компонент `Button` имеет восемь маркеров, а компонент `CheckBox` только один.

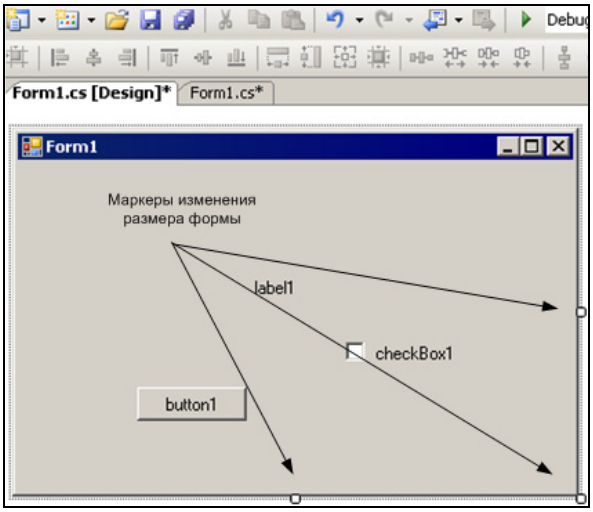


Рис. 2.2. Редактор формы

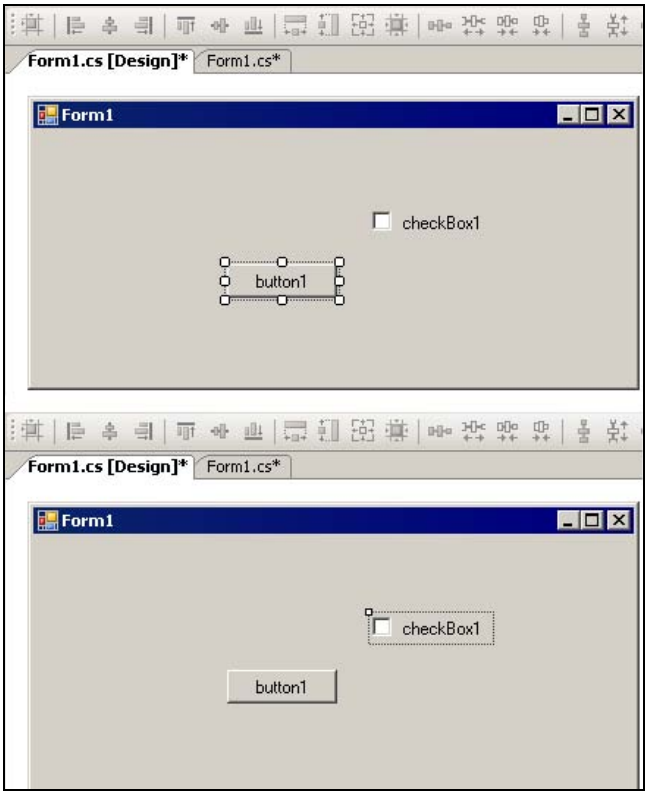


Рис. 2.3. Маркеры изменения размеров компонентов

Если при выборе компонентов на форме удерживать клавишу <Shift>, можно выбрать несколько компонентов одновременно. Первый из них будет иметь белые маркеры и являться *основным компонентом* (primary) в выбранной группе. Остальные выбранные компоненты будут иметь темные маркеры (рис. 2.4).

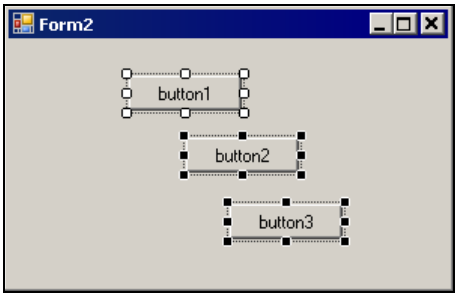


Рис. 2.4. Маркеры первого в группе компонента имеют белый цвет

Интересной особенностью редактора Visual Studio является подсказка о взаимном расположении элементов (snap lines). Например, при перемещении кнопки `button1` вправо редактор отображает синюю маркерную линию, ведущую к компоненту `Checkbox1`, которая помогает выполнить выравнивание компонента вправо (рис. 2.5).

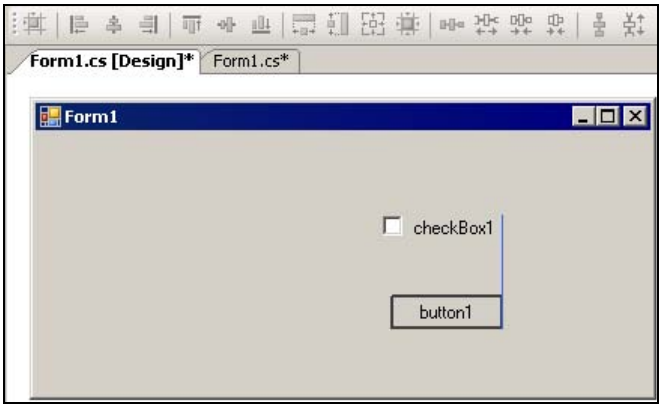


Рис. 2.5. Выравнивание взаимного расположения компонентов формы

2.3. Визуальные и невизуальные компоненты в дизайнере формы

Библиотека .NET Framework имеет два типа компонентов: *визуальные* и *невизуальные*. Визуальные компоненты являются элементами пользовательского интерфейса. Это, например, компоненты: кнопка (`Button`), выпадающий список (`ComboBox`) или метка (`Label`). Невизуальные компоненты не имеют пользовательского интерфейса и не могут располагаться на форме. Дизайнер Visual Studio располагает их внизу окна дизайнера[1] (рис. 2.6). Такими компонентами являются, например, компоненты работы с базами данных, таймер (`Timer`) и компонент работы с последовательным портом (`SerialPort`).

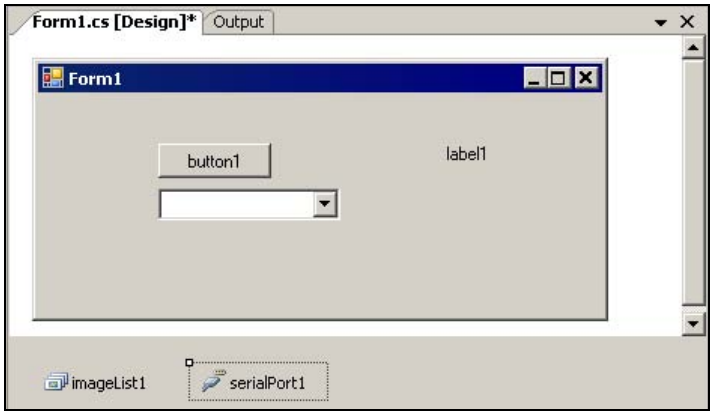


Рис. 2.6. Визуальные и невизуальные компоненты

2.4. Палитра компонентов

Окно **Toolbox** (Инструменты) во время редактирования формы превращается в набор компонентов (рис. 2.7). Название окна, которое пришло из Visual Basic, при этом никак не меняется, что, мне кажется, не совсем удачно. Правильнее было бы назвать это окно "палитрой компонентов", как это было в Delphi. С другой стороны, сюда же вынесли содержимое буфера обмена[2], так что и на чистый набор компонентов это тоже не очень похоже. Давайте

[1] Для дотошного читателя могу сказать, что для таких компонентов создается специальный контейнер, имеющий тип `System.Windows.Forms.Design.ComponentTray`.

[2] К нашему обсуждению это не относится, но все-таки замечу, что окно **Toolbox** (Инструменты) в Visual Studio можно использовать как многопозиционный буфер обмена. Попробуйте перетащить текст из редактора кода на это окно и обратно.

договоримся, что содержимое окна **Toolbox** (Инструменты), когда мы находимся в редакторе форм и видим набор компонентов, мы будем называть *палитрой компонентов*. Так будет более понятно.

Компоненты в палитре компонентов объединяются в *категории* (categories), которые позволяют быстро найти нужные элементы. Например, категория **Menus&Toolbars** объединяет компоненты для построения меню и панелей инструментов, а категория **Data** — компоненты для работы с данными.

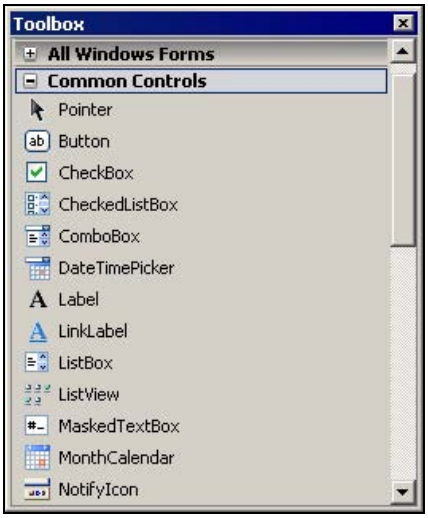


Рис. 2.7. Окно **Toolbox**

2.5. Окно свойств

Меню **Properties** (Свойства) открывает окно свойств выбранного в текущий момент объекта (рис. 2.8), который можно изменить, выбрав нужный объект в верхнем выпадающем списке. Стандартное окно свойств содержит две закладки[1], переключаемые кнопками  и  . Первая кнопка переключает на закладку свойств объекта. Здесь собраны все свойства, которые позволяют посмотреть и изменить выбранный объект. Вторая кнопка переключает на закладку **Events** (События), которая содержит список событий, доступных для выбранного объекта.

Именно свойства и события являются наиболее важными интерфейсами взаимодействия. Чем удачнее подобран набор свойств, тем больше вероят-

[1] В *разд. 12.3* я расскажу как добавлять собственные закладки.

ность, что конечные пользователи (такие же программисты, как и мы) останутся довольны. С этой точки зрения лучше сделать больше свойств, чем меньше. Но, конечно, переусердствовать тоже не стоит, иначе пользователь компонента просто запутается. Хорошим подходом также является разработка собственного редактора свойств, позволяющего более удобно (чем обычные редакторы) редактировать все нужные свойства. Еще один вариант — объединение свойств в сложные свойства, состоящие из других свойств. Например, сложное свойство `Font` состоит из нескольких простых свойств. При этом редактировать можно как само свойство `Font`, так и каждую из его составляющих.

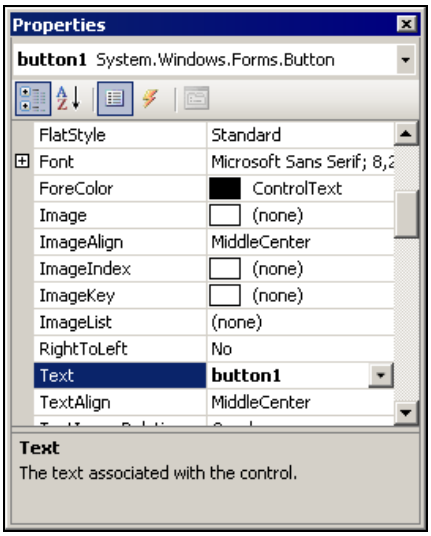


Рис. 2.8. Окно **Properties**

2.6. Редакторы свойств

Каждое свойство, в зависимости от его типа, имеет свой *редактор свойств*. Среда VS предоставляет редакторы для всех стандартных свойств, которые показаны на рис. 2.9—2.15:

- простой редактор текстовых и числовых полей позволяет изменять все свойства с типом `string`, `int` и т. п.;
- редактор выбора из списка позволяет модифицировать перечисления (`enum`), а также свойства типа `bool` (выбор из списка `true/false`);
- редактор цветов позволяет редактировать свойства типа `Color`;

- редактор шрифтов позволяет изменять шрифт с помощью специального диалога или каждую из составляющих;
- редакторы свойств `TextAlign` и `Dock` показывают специальные диалоговые окна.

В дальнейшем мы научимся создавать собственные редакторы свойств.

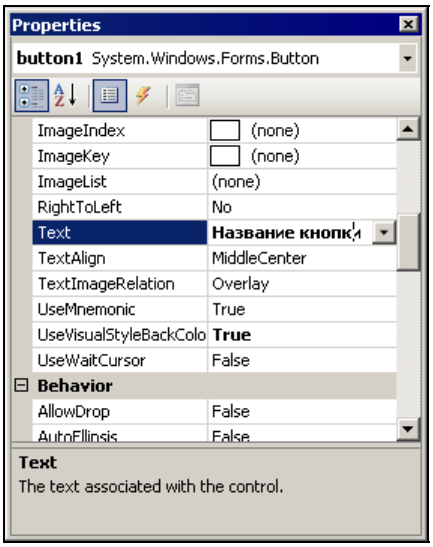


Рис. 2.9. Редактор простого (текстового или числового) поля

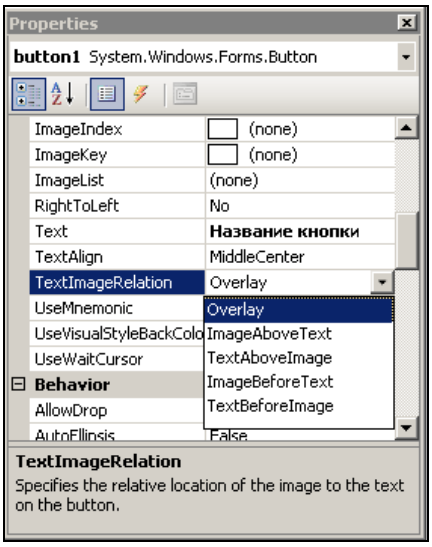


Рис. 2.10. Редактор с помощью выбора из списка

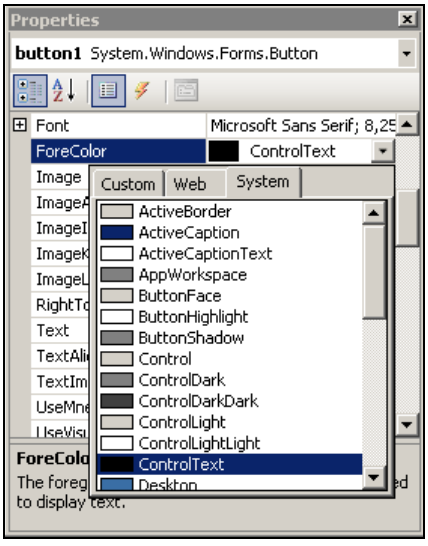


Рис. 2.11. Редактор цветов

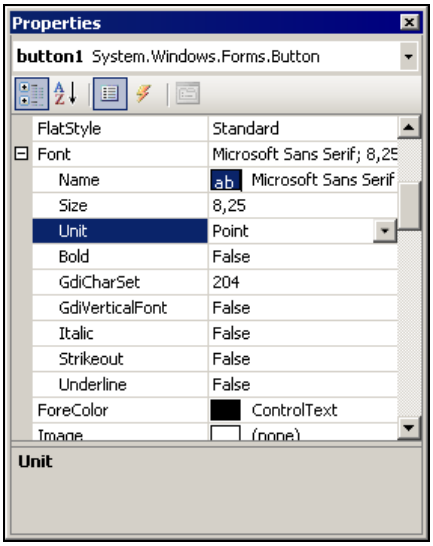


Рис. 2.12. Редактор шрифта (по частям)

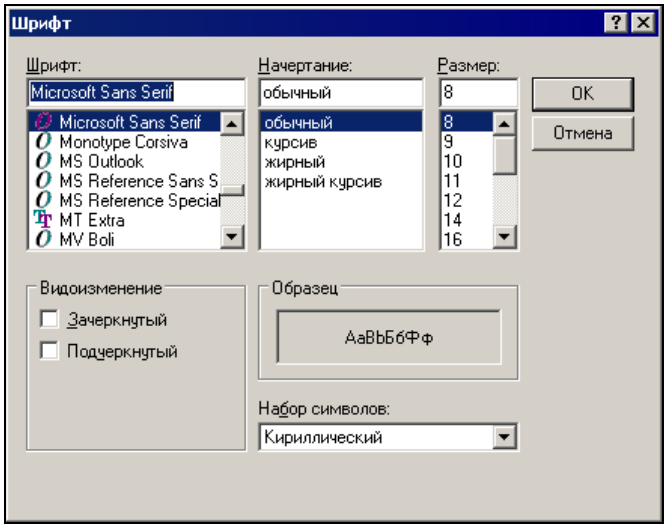


Рис. 2.13. Редактор шрифта

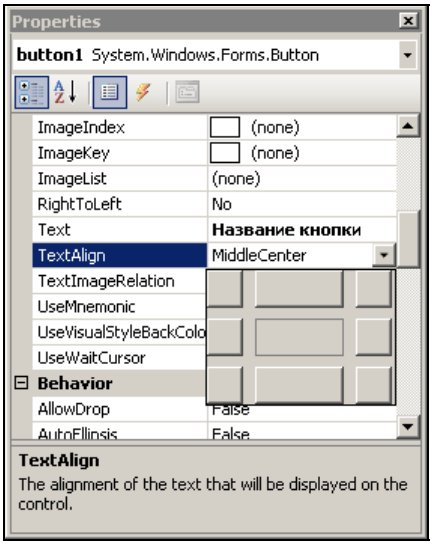


Рис. 2.14. Редактор выравнивания (свойство `TextAlign`)

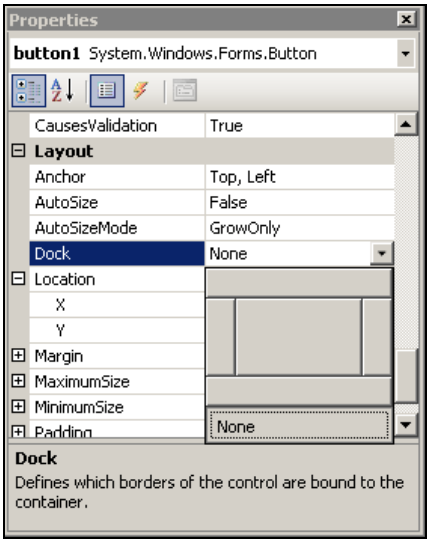
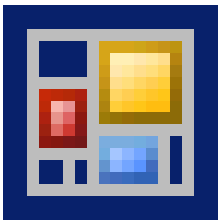


Рис. 2.15. Редактор привязки (свойство `Dock`)

ГЛАВА 3



Немного теории

> Сложные проблемы всегда имеют простые, легкие для понимания неправильные решения.

Эта глава содержит тот минимум теории, который будет необходим для дальнейшего рассказа. Начинающим я рекомендую обязательно прочитать эту главу. Тем, кто знаком с синтаксисом языка, но не сталкивался с методами отражения и метаинформацией, следует начать с *разд. 3.3*. Если же вы хорошо знакомы с языком C#, можете смело пропустить эту главу и читать дальше.

3.1. Свойства и события

Любой компонент взаимодействует с внешним миром с помощью свойств и событий. *Свойства* (properties) позволяют настраивать и получать параметры компонента, а *события* (events) — реагировать на происходящие внутри компонента события, программировать реакцию компонента на внешние события и т. д.

С точки зрения кода, события — сокращенная запись методов. Рассмотрим, например, такой код:

```
class Test
{
  private int testInt;

  public int TestInt
  {
    get { return testInt;  }
```

```
    set { testInt = value; }
  }
}
```

Этот фрагмент можно было бы написать так:

```
class Test
{
  private int testInt;

  public int GetTestInt()
  {
    return testInt;
  }

  public void SetTestInt(int value)
  {
    testInt = value;
  }
}
```

Таким образом, запись в виде свойства уменьшает количество кода, оставляя основное преимущество свойств — возможность дополнительных действий при получении или записи значения поля. Действительно, доступ к обычному полю не дает никакой возможности, например, проверить значение на правильность. Сравните два фрагмента кода:

```
// Доступ к полю
class Test
{
  public int testInt;
}
int v = testInt;

// Доступ к свойству
class Test
{
  private int testInt;

  public int TestInt
  {
    get { return testInt; }
    set {
        if (testInt < 0)
           throw new ArgumentOutOfRangeException("TestInt");
```

```
        testInt = value;
    }
  }
}
```

Дополнительная проверка во время установки значения свойства позволяет отсечь некорректные значения. Аналогичный код для поля нужно было бы писать во всех местах, где производится установка значения этого поля.

Методы `get` и `set`, которые, по сути, являются оболочкой поля, называют *аксессорами* (accessor). Еще одно преимущество свойств — возможность задавать разный уровень доступа аксессоров:

```
class Test
{
  private int testInt;

  public int TestInt
  {
    get { return testInt;  }
    protected set { testInt = value; }
  }
}
```

Здесь, аксессор `get` получает уровень видимости `public`, тогда как `set` ограничивается уровнем `protected`. Извне такое свойство будет выглядеть как свойство "только для чтения".

Кажется, информации о свойствах у нас вполне достаточно, перейдем к событиям.

Модель *событий* (event) в .NET основана на *делегатах* (delegate). Делегаты позволяют обращаться к *методам обратного вызова* (callback method). Метод обратного вызова — это механизм, позволяющий объекту получать уведомления, на которые он подписался.

Описание события производится с помощью специального слова `event`:

```
public class TestClass
{
  private event EventHandler OnChange;
  public  event EventHandler OnChange
  {
    add    {  onChange += value; }
    remove {  onChange -= value; }
  }
}
```

По виду записи события похожи на свойства. Метод `add` позволяет подписать объект на событие, а метод `remove` — удалить подписку.

Тот же самый код будет сгенерирован, если описать событие с модификатором `public`:

```
public class TestClass
{
  public event EventHandler OnChange;
}
```

Методы `add`, `remove` и закрытое поле делегата будут сгенерированы компилятором автоматически.

Когда компоненту нужно вызвать событие, используется следующий код:

```
if (onChange != null)
  onChange(this, new EventArgs());
```

Проверка на `null` позволяет обработать ситуацию, когда обработчик события не задан.

Класс `EventHandler` описывает тип события. Его определение выглядит так:

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

Можно сказать, что по умолчанию (т. е. при использовании типа `EventHandler`) у события будут два аргумента. Параметр *`sender`* определяет "владельца" события, и поэтому при вызове события обычно передается значение `this`. Второй параметр представляет собой пустой аргумент, описываемый с помощью класса `EventArgs`:

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

Конечно, параметры события не всегда столь примитивны. При необходимости передать больше информации в параметрах нужно создать собственный наследник класса `EventArgs` и описать соответствующий тип параметра:

```
// Описание аргумента с параметром Name
public class ChangeArgs : EventArgs
{
  private readonly string name;
```

```
  public ChangeArgs(string name)
  {
    this.name = name;
  }

  public string Name
  {
    get { return name; }
  }
}

// Описание события
public event EventHandler<ChangeArgs> OnChange;
  // Вызов события
  if (OnChange != null)
     OnChange(this, new ChangeArgs(this.Name));
```

Сам обработчик события будет выглядеть, например, так:

```
private void gradientLabel1_OnChange(object sender,
                                     MyControl.ChangeArgs e)
{
  // Обработка
}
```

Реализация методов `add` и `remove` самостоятельно нужна, например, если необходимо создать потокобезопасный код (как я уже говорил, по умолчанию можно оставить работу по генерации этих методов компилятору):

```
// Описание объекта синхронизации
private readonly object eventLock = new object();
// Закрытый делегат
private EventHandler<ChangeArgs> onChange;

// Описание события (методы add и remove)
public event EventHandler<ChangeArgs> OnChange
{
  add
  {
    lock (eventLock)
    {
      onChange += value;
    }
  }
```

```
  remove
  {
    lock (eventLock)
    {
      onChange -= value;
    }
  }
}

// Вызов события
if (onChange != null)
  onChange(this, new ChangeArgs(this.Name));
```

Это все, что я хотел рассказать о свойствах и событиях. Конечно, я понимаю, что это очень мало, но, согласитесь, что эту информацию вполне можно найти во многих источниках, например в MSDN или в [5].

3.2. Интерфейсы

Интерфейс — это именованный набор сигнатур методов (см. [5]). Интерфейсы могут определять события и свойства, но не могут определять методов-конструкторов и экземплярных полей.

В C# для определения интерфейса используется ключевое слово `inferface`, например:

```
public interface IDisposable
{
  void Dispose();
}

public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Любой класс, наследующий (правильнее сказать, реализующий) интерфейс должен реализовать все сигнатуры, описанные в этом интерфейсе:

```
public class Test : IDisposable
{
  public void Dispose() { Console.WriteLine("Вызов Dispose"); }
}
```

В C# класс может реализовывать несколько интерфейсов, а также интерфейс может наследовать другие интерфейсы.