Урок №4

Содержание

1. Печать в Windows Forms

За печать в Windows Forms отвечают несколько взаимосвязанных классов, которые, ссылаются друг на друга. Так, у класса **PrinterSettings** есть свойство типа **PageSettings,** а у класса **PageSettings -** свойство типа **PrinterSettings.** При изучении печати по большей части приходится заниматься отбором нужных классов.

Пространство имен **System.Windows.Forms** содержит классы, способные выводить стандартные диалоговые окна печати, параметры страницы и предварительного просмотра документа.

Принтеры и их параметры (класс PrinterSettings)

Windows разрешает пользователю устанавливать несколько принтеров, а точнее, **драйверов** для нескольких принтеров. В любой момент времени один из принтеров, перечисленных в списке установленных принтеров, является **принтером по умолчанию.** Принтер по умолчанию можно выбрать в диалоговом окне Printers.

Для программы Windows Forms описанием принтера является объект типа **PrinterSettings.** Подобно большинству объектов, о которых пойдет речь в этой части урока (кроме стандартных диалоговых окон), он определяется в пространстве имен **System.Drawing.Printing.** У класса **PrinterSettings** есть только конструктор по умолчанию, который создает объект **PrinterSettings** для принтера по умолчанию.

Конструктор *PrinterSettings*

```
PrinterSettings();
```

Например, оператор:

```
PrinterSettings prnset = new PrinterSettings();
```

создает новый экземпляр объекта **PrinterSettings,** ссылающийся на принтер по умолчанию.

Следующие три свойства предоставляют некоторые базовые сведения о принтере:

Свойства *PrinterSettings* (выборочно)

Тип	**Свойство**	**Доступ**
string	*PrinterName*	Чтение/запись
bool	*IsValid*	Чтение
bool	*IsDefaultPrinter*	Чтение

PrinterName - это строка, обычно содержащая имя и модель принтера, такая же строка выводится в диалоговом окне Printers.

Устанавливая принтер, пользователь может изменить его имя, поэтому имя принтера, показанное объектом **PrinterSettings**, может отличаться от стандартного.

При создании нового объекта **PrinterSettings** свойствам **IsValid** и **IsDefaultPrinter** обычно присваивается значение **true.** Но если в системе не установлен ни один принтер, свойство **PrinterName** возвращает строку «<no default printer>», а **IsValid** равно **false.**

Свойство **PrinterName** изменяемое. Это означает, что если в него занести строку, идентифицирующую установленный принтер, то в остальные свойства объекта **PrinterSettings** также будут занесены значения для этого принтера. Ясно, что строка, заносимая в это свойство, должна совпадать с именем одного из установленных принтеров. Хотя в противном случае исключения не возникает, свойству **IsValid** будет присвоено значение **false.**

Чтобы не сделать ошибки при изменении свойства **PrinterName,** сначала нужно получить список установленных принтеров. Для этого служит единственное статическое свойство объекта **PrinterSettings:**

Статическое свойство *PrinterSettings*

Тип	**Свойство**	**Доступ**
PrinterSettings.StringCollection	*InstalledPrinters*	Чтение

Класс **StringCollection** определяется в классе **PrinterSettings,** реально это всего лишь массив неизменяемых строк. Допустим, свойство **InstalledPrinters** использовано так:

```
PrinterSettings.StringCollection sc = PrinterSettings.InstalledPrinters();
```

После этого будут доступны два свойства объекта **sc**

Свойства *PrinterSettings.StringCollection*

Тип	**Свойство**	**Доступ**
int	*Count*	Чтение
String	*[]*	Чтение

Значение **sc.Count** - это число установленных принтеров. Если установленных принтеров нет, оно равно нулю. Таким образом, **sc[0]-** это имя первого принтера, **sc[1] -** второго и т. д.

Можно не сохранять значение **InstalledPrinters** в переменной, а обращаться к самому свойству. Так, значение:

```
PrinterSettings.InstalledPrinters.Count;
```

- это число установленных принтеров, а

```
PrinterSettings.InstalledPrinters[1];
```

- имя второго принтера. Следующий код помещает имена всех установленных принтеров в объект **ComboBox** с именем **combo:**

```
foreach (string str in PrinterSettings.InstalledPrinters)
                combo.Items.Add(str);
```

Можно изменить принтер, на который ссылается объект **PrinterSettings,** поместив в свойство **PrinterName** строку из набора **StringCollection.** Если **sc -** переменная для хранения **StringCollection,** это делается так:

```
prnset.PrinterName = sc[2];
```

Можно присвоить значение **PrinterName** напрямую, проиндексировав свойство **InstalledPrinters:**

```
prnset.PrinterName = PrinterSettings.InstalledPrinters[2]
```

Если не возникнет серьезных ошибок, свойству **IsValid** в результате должно быть присвоено **true,** а **IsDefaultPrinter -false** (даже если занести в **PrinterName** имя принтера по умолчанию).

Вот пара свойств, описывающих фундаментальные возможности принтера:

Свойства PrinterSettings (выборочно)

Тип	**Свойство**	**Доступ**

bool	*IsPlotter*	Чтение
bool	*SupportsColor*	Чтение
int	*LandscapeAngle*	Чтение

Если свойство **IsPlotter** равно **true,** растровые изображения на таком принтере печатать невозможно. Второе свойство говорит о том может ли принтер поддерживать цветную печать. Свойство **LandscapeAngle** обычно равно 90° или 270°, но если принтер не поддерживает печать страниц в альбомной ориентации, оно будет равно нулю.

Объект **PrinterSettings** ничего не говорит о технологии печати принтера (лазерная она, струйная или иная).

Три свойства объекта **PrinterSettings** возвращают наборы элементов и показывают доступные источники бумаги (лотки), перечень поддерживаемых принтером размеров бумаги (включая размеры конвертов) и список доступных разрешений:

Свойства *PrinterSettings* (выборочно)

Тип	**Свойство**	**Доступ**	**Элементы**
PrinterSettings.PaperSourceCollection	*PaperSources*	Чтение	*PaperSource*
PrinterSettings.PaperSizeCollection	*PaperSizes*	Чтение	*PaperSize*
PrinterSettings.PrinterResolutionCollection	*PrinterResolution*	Чтение	*PrinterResolution*

Принцип работы этих свойств довольно прост. Все три класса (столбец Тип) определяются в классе **PrinterSettings.** У каждого есть по паре неизменяемых свойств: **Count,** соответствующее числу элементов набора, и индексатор, возвращающий объект, тип которого указан в столбце Элементы.

Например, свойство **PaperSources** по существу является набором объектов **PaperSource.** Значение:

```
prnset.PaperSources.Count;
```

указывает число этих объектов в наборе. Индексируя свойство, можно ссылаться на любой из них. Таким образом:

```
prnset.PaperSources[2];
```

это третий объект типа **PaperSource** из набора **PrinterSettings.PaperSourceCollection.**

Рассмотрим классы **PaperSource, PaperSize** и **PrinterResolution.** У класса **PaperSource** всего два свойства, оба неизменяемые: **SourceName** типа **string** и **Kind** типа **PaperSourceKind.** Свойство **SourceName** содержит текст описания, имеющего значение для пользователя (например, «ручная подача бумаги»), а **PaperSourceKind -** это перечисление. Свойство **PaperSources** объекта **PrinterSettings -** это **полный** набор источников бумаги принтера. Его значение **не** отражает источник бумаги, выбранный в данный момент как источник бумаги по умолчанию.

Свойство **PaperSizes** объекта **PrinterSettings -** это полный набор размеров бумаги, поддерживаемых принтером. Каждый элемент этого набора - объект типа

PaperSize с четырьмя свойствами: **PaperName, Width, Height, Kind. PaperName -** это текстовая строка, имеющая значение для пользователя. Свойства **Width** и **Height** указывают размер бумаги (конверта) в сотых долях дюйма. **PaperKind -** это перечисление, включающее более сотни членов, описывает типы бумаги.

Свойство **PrinterResolutions** объекта **PrinterSettings -** это набор **объектов PrinterResolution,** которые показывают реальные значения разрешения устройства, доступные на принтере. Разрешение указывается в точках на дюйм.

Следующее свойство класса **PrinterSettings -** это объект **DefaultPageSettings** типа **PageSettings,** который представляет еще один важный класс, определяемый в пространстве имен **System.Drawing.Printing. К**ласс **PageSettings** описывает характеристики печатаемой страницы. Так, у объекта **PrinterSettings** есть свойство **PaperSources,** содержащее набор всех доступных источников бумаги принтера. Свойство **PaperSource** объекта **PageSettings** указывает источник бумаги для некоторой страницы. Свойство **DefaultPageSettings** объекта **PrinterSettings** определяет параметры страницы по умолчанию. При печати можно менять параметры страницы как для всего документа, так и для любой из его страниц по отдельности.

Метод объекта **PrinterSettings** под названием **CreateMeasurementGraphics()** возвращает то, что в Win32 называется **контекстом устройства** (information device context). Объект **Graphics,** полученный при помощи **CreateMeasurementGraphics**, позволяет не только рисовать на странице принтера, но и получать сведения о нем. Этот метод в любое время (например, во время исполнения конструктора программы) выдаст дополнительные сведения о любом из установленных принтеров.

Параметры страницы (Класс PageSettings)

Класс **PageSettings** описывает параметры принтера, которые могут меняться от страницы к странице. Однако любой объект **PageSettings** всегда связан с принтером, так как если объект **PageSettings** указывает, что страница должна быть напечатана на бумаге определенного формата, то необходимо, чтобы принтер поддерживал именно этот формат.

Обычно программы обращаются к готовым объектам **PageSettings** (например, к свойству **DefaultPageSettings** объекта **PrinterSettings,** которое содержит такой объект), но конструктор этого класса также позволяет создавать новые объекты **PageSettings.**

Конструкторы *PageSettings*

```
PageSettings();
PageSettings(PrinterSettings prnset);
```

Первый конструктор создает объект **PageSettings** для принтера по умолчанию, а второй - для любого из установленных принтеров, заданного аргументом **PrinterSettings.** В обоих случаях созданный объект **PageSettings** содержит параметры страницы по умолчанию для заданного принтера.

Пользователи определяют параметры страницы по умолчанию для установленных принтеров в диалоговом окне со свойствами данного принтера (PrintingPreferences), которое можно вызвать из диалогового окна Printers. Windows Forms может изменить эти

параметры во время печати документа, но никакие изменения, внесенные Windows Forms, не повлияют на другие приложения.

У класса **PageSettings** восемь свойств, семь из которых изменяемы:

Свойства PageSettings

Тип	**Свойство**	**Доступ**
PrinterSettings	*PrinterSettings*	Чтение/запись
bool	*Landscape*	Чтение/запись
Rectangle	*Bounds*	Чтение
Margins	*Margins*	Чтение/запись
bool	*Color*	Чтение/запись
PaperSource	*PaperSource*	Чтение/запись
PaperSize	*PaperSize*	Чтение/запись
PrinterResolution	*PrinterResolution*	Чтение/запись

Первое свойство **(PrinterSettings)** задает принтер, с которым ассоциированы эти параметры страницы. Если получить объект **PrinterSettings** из свойства **DefaultPageSettings** объекта **DefaultPageSettings,** у объекта **PageSettings** в свойстве с именем **PrinterSettings** окажется объект, тождественный исходному **PrinterSettings.**

Остальные свойства **PageSettings** обычно используются просто для получения нужных сведений. Однако программа может изменять значения этих свойств (в определенных переделах), чтобы изменить способ печати страницы. Так, свойство **Landscape** позволяет получить сведения об ориентации страницы: **false** указывает, что задана книжная ориентация страницы, а **true -** что альбомная. Если приложение должно печатать разные документы в зависимости от ориентации страницы, эти сведения позволяют выбрать, что печатать. Программа также может самостоятельно изменять это свойство без вмешательства пользователя.

Неизменяемое свойство **Bounds -** это объект **Rectangle,** который указывает размер страницы с учетом размера бумаги и значения свойства **Landscape,** выраженный в сотых долях дюйма. Свойство **Margins** указывает ширину полей страницы по умолчанию. Исходно для всех полей страницы установлена ширина в 1 дюйм. Новый объект **Margins** создается с помощью конструкторов:

Конструкторы *Margins*

```
Margins();
Margins(int Left, int Right, int Top, int Bottom);
```

Четыре свойства этого класса указывают ширину полей в сотых долях дюйма.

Иногда пользователь запрещает печатать страницу в цвете, хотя принтер поддерживает цветную печать. Свойство **Color** объекта **PageSettings** указывает, хочет ли пользователь печатать страницу в цвете.

Три свойства объекта **PageSettings: PaperSource**, **PaperSize** и **PrinterResolution** - соответствуют трем свойствам класса **PrinterSettings**: **PaperSources, PaperSizes** и **PrinterResolutions,** с которыми вы уже знакомы (обратите внимание на форму множественного числа всех имен свойств класса **PrinterSettings).** Так, свойство **PaperSource** объекта **PageSettings** - это один из элементов набора **PaperSources,** составляющего одноименное свойство объекта **PrinterSettings.**

Значение свойства **Landscape** не влияет на свойство **PaperSize.** Если **Landscape** равно **false,** то значения **Width** и **Height** у **Bounds** равны соответствующим значениям свойства **PaperSize,** иначе значения свойств **Width** и **Height** взаимозаменяются, тогда как с аналогичными свойствами **PaperSize** этого не происходит.

Определение документа для печати (класс PrintDocument)

Задание печати состоит из одной или нескольких страниц, печатаемых на принтере. Оно представлено объектом **PrintDocument,** у которого есть лишь конструктор по умолчанию:

Конструктор *PrintDocument*

```
PrintDocument();
```

В общем случае программа начинает процесс печати с создания объекта типа **PrintDocument:**

```
PrintDocument prndoc = new PrintDocument();
```

Для каждого задания печати можно создать новый объект **PrintDocument.** Если же пользователь выбирает принтер и устанавливает его параметры в стандартном диалоговом окне печати (или при помощи других средств), лучше сохранять заданные параметры в объекте **PrintDocument** и применять этот экземпляр объекта в течение всего сеанса работы программы. В этом случае объект **prndoc** определяется как поле и создается однократно.

У **PrintDocument** всего четыре свойства, но два содержат объекты типа **PrinterSettings** и **PageSettings,** так что в этом объекте заключено намного больше информации, чем может показаться:

Свойства *PrintDocument*

Тип	**Свойство**	**Доступ**
PrinterSettings	*PrinterSettings*	Чтение/запись
PageSettings	*DefaultPageSettings*	Чтение/запись
string	*DocumentName*	Чтение/запись
PrintController	*PrintController*	Чтение/запись

При создании нового объекта **PrintDocument** свойство **PrinterSettings** указывает принтер по умолчанию. При желании можно изменять свойство **PrinterSettings** или его отдельные свойства.

Первоначально свойство **DefaultPageSettings** устанавливают на основе значения свойства **DefaultPageSettings** объекта **PrinterSettings.** Его тоже можно изменять, как и значения отдельных свойств этого свойства.

Свойство **DocumentName** инициализируется текстовой строкой «document», которую, скорее всего, потребуется изменить. Это отображаемое имя задания печати, которое всегда используется для его идентификации. Свойство **PrintController** рассматривается далее.

У класса **PrintDocument** четыре открытых события:

События *PrintDocument*

Событие	**Метод**	**Делегат**	**Аргумент**
BeginPrint	*OnBeginPrint*	*PrintEventHandler*	*PrintEventArgs*
QueryPageSettings	*OnQueryPage Settings*	*QueryPageSettings EventHandler*	*QueryPageSettingsEventArgs*
PrintPage	*OnPrintPage*	*PrintPageEvent Handler*	*PrintPageEventArgs*
EndPrint	*OnEndPrint*	*PrintEventHandler*	*PrintEventArgs*

События **BeginPrint** и **EndPrint** происходят однократно для каждого задания печати, а **QueryPageSettings** и **PrintPage** - для каждой страницы задания. Обработчик события **PrintPage** указывает, остались ли еще страницы, которые должны быть напечатаны.

Как минимум, устанавливается обработчик для **события PrintPage.** Если для каждой страницы нужно задавать собственные параметры, придется также установить обработчик для события **QueryPageSettings.** Чтобы выполнить длинные процедуры инициализации или освобождения ресурсов, устанавливают обработчики для событий **BeginPrint** и **EndPrint.**

Чтобы инициировать печать, нужно вызвать следующий метод объекта **PrintDocument,** не связанный ни с одним событием:

Метод *PrintDocument*

```
void Print();
```

Метод **Print** не возвращает управление, пока программа не закончит печать документа. В это время приложение не реагирует на действия пользователя. Во время исполнения метода **Print** вызываются обработчики события **PrintDocument,** подключенные этой программой. Сначала вызывается обработчик **BeginPrint,** затем - **QueryPageSettings** и **PrintPage** для каждой страницы и, наконец, - **EndPrint**.

Обработка событий объекта PrintDocument

Объект **CancelEventArgs** определяется в пространстве имен **System.ComponentModel.** Объект **PrintEventArgs,** связанный с событиями **BeginPrint** и **EndPrint,** обладает единственным свойством, унаследованным от **CancelEventArgs:**

Свойство *PrintEventArgs*

Тип	Свойство	Доступ
bool	*Cancel*	Чтение/запись

Обработчик события **BeginPrint** может присвоить свойству **Cancel** значение **true,** чтобы прервать исполнение задания печати (скажем, если задание требует больше памяти, чем доступно в системе).

Класс **QueryPageSettingsEventArgs** добавляет к **Cancel** еще одно свойство:

Свойство *QueryPageSettingsEventArgs*

Тип	Свойство	Доступ
PageSettings	*PageSettings*	Чтение/запись

Обработчик события **QueryPageSettings** может изменять свойства объекта **PageSettings,** чтобы подготовиться к соответствующему событию **PrintPage.**

У класса **PrintPageEventArgs** четыре неизменяемых свойства и два изменяемых:

Свойства *PrintPageEventArgs*

Тип	Свойство	Доступ
Graphics	*Graphics*	Чтение
bool	*HasMorePages*	Чтение/запись
bool	*Cancel*	Чтение/запись
PageSettings	*PageSettings*	Чтение
Rectangle	*PageBounds*	Чтение
Rectangle	*MarginBounds*	Чтение

Для печати каждой страницы объект **Graphics** создается заново. Установленные свойства объекта **Graphics** для одной страницы (например, **PageUnit** или **PageScale)**, не будут действовать на следующие страницы. По умолчанию свойство **PageUnit** равно **GraphicsUnitDisplay,** поэтому принтер кажется устройством с разрешением 100 dpi. Свойства **DpiX и DpiY объекта Graphics** отражают значение свойства **PrinterResolution** объекта **PageSettings.**

На входе в обработчик события печати страницы свойству **HasMorePages** всегда присваивается **false.** Если печатается многостраничный документ, по завершении обработчика события нужно присвоить этому свойству **true,** чтобы обработчик был

вызван снова для печати следующей страницы. По завершении печати последней страницы следует оставить это свойство равным **false.** Свойству **Cancel** обычно задается **false.** Если программе нужно прервать исполнение задания печати, этому свойству надо присвоить **true.** Задать свойству **Cancel** значение **true** и оставить **true** свойству **HasMorePages -** далеко не одно и то же: в первом случае ОС попытается остановить печать страниц, уже поставленных в очередь печати.

Свойство **PageSettings** служит для получения сведений о странице во время печати. Оно отражает все изменения параметров страницы, которые делаются при исполнении обработчика события **QueryPageSettings.**

Для удобства программиста в объект **PrintPageEventArgs** включен прямоугольник **PageBounds,** тождественный значению свойства **Bounds** объекта **PageSettings,** и **MarginBounds,** представляющий размеры страницы за вычетом полей, ширина которых задана свойством **Margins** объекта **PageSettings.**

Следующая программа выводит простое диалоговое окно, позволяющее пользователю выбрать принтер из списка установленных принтеров.

```
//
// PrinterSelectisnDialog.cs
//
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

namespace PrinterSelectionDialog
{
    class PrinterSelectionDialog : Form
    {
        ComboBox combo;

        public PrinterSelectionDialog()
        {
            Text = "Select Printer";
            FormBorderStyle = FormBorderStyle.FixedDialog;
            ControlBox = false;
            MaximizeBox = false;
            MinimizeBox = false;
            ShowInTaskbar = false;
            StartPosition = FormStartPosition.Manual;
            Location = ActiveForm.Location +
                       SystemInformation.CaptionButtonSize +
```

```
                    SystemInformation.FrameBorderSize;

            Label label = new Label();
            label.Parent = this;
            label.Text = "Printer;";
            label.Location = new Point(8, 8);
            label.Size = new Size(40, 8);

            combo = new ComboBox();
            combo.Parent = this;
            combo.DropDownStyle = ComboBoxStyle.DropDownList;
            combo.Location = new Point(48, 8);
            combo.Size = new Size(144, 8);

            // Добавить установленные принтеры в раскрывающийся список.

            foreach (string str in PrinterSettings.InstalledPrinters)
                combo.Items.Add(str);

            Button btn = new Button();
            btn.Parent = this;
            btn.Text = "OK";
            btn.Location = new Point(40, 32);
            btn.Size = new Size(40, 16);
            btn.DialogResult = DialogResult.OK;
            AcceptButton = btn;

            btn = new Button();
            btn.Parent = this;
            btn.Text = "Cancel";
            btn.Location = new Point(120, 32);
            btn.Size = new Size(40, 16);
            btn.DialogResult = DialogResult.Cancel;

            CancelButton = btn;

            ClientSize = new Size(200, 56);
            AutoScaleBaseSize = new Size(4, 8);
        }
```

```
        public string PrinterName
        {
            set { combo.SelectedItem = value; }
            get { return (string)combo.SelectedItem; }
        }
    }
}
```

Список относится к типу **DropDownList,** поэтому пользователю запрещено вводить что-либо в поле редактирования. Изменяемое свойство **PrinterName** позволяет программе инициализировать выбранный элемент списка именем принтера по умолчанию и получить выбранный пользователем элемент.

Такое же диалоговое окно вы видите и в программе PrintThreePages. Чтобы пользователь смог инициировать печать, она создает меню с вложенным меню File и командой Print. Обработчик элемента меню Print выводит диалоговое окно, где пользователь может выбрать принтер. Программа подключает обработчики событий **QueryPageSettings** и **PrintPage,** чтобы напечатать три страницы. В центре каждой страницы крупным шрифтом написан ее номер. Программа устанавливает разрешение «draft» для всего документа и попеременно назначает для страниц то книжную, то альбомную ориентацию:

```
//
// PrintThreePages.cs
//
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

namespace PrintThreePages
{
    class PrintThreePages : Form
    {
        const int iNumberPages = 3;
        int iPageNNumber;

        public PrintThreePages()
        {
            Text = "Print Three Pages";
            Menu = new MainMenu();
            Menu.MenuItems.Add("&File");
```

```
            Menu.MenuItems[0].MenuItems.Add("&Print...",
                new EventHandler(MenuFilePrintOnClick));
        }

        void MenuFilePrintOnClick(object obj, EventArgs ea)
        {
            // Create PrintDocument.
            PrintDocument prndoc = new PrintDocument();

            // Create dialog box and set PrinterName property.
            PrinterSelectionDialog dlg = new PrinterSelectionDialog();
            dlg.PrinterName = prndoc.PrinterSettings.PrinterName;

            // Show dialog box and bail out if not OK.
            if (dlg.ShowDialog() != DialogResult.OK)
                return;

            // Set PrintDocument to selected printer.
            prndoc.PrinterSettings.PrinterName = dlg.PrinterName;

            // Set printer resolution to "draft".
            foreach (PrinterResolution prnres in
                prndoc.PrinterSettings.PrinterResolutions)
            {
                if (prnres.Kind == PrinterResolutionKind.Draft)
                    prndoc.DefaultPageSettings.PrinterResolution = prnres;
            }

            // Set remainder of PrintDocument properties.
            prndoc.DocumentName = Text;
            prndoc.PrintPage += new PrintPageEventHandler(OnPrintPage);
            prndoc.QueryPageSettings +=
                new QueryPageSettingsEventHandler(OnQueryPageSettings);

            // Commence printing.
            iPageNNumber = 1; prndoc.Print();
        }

        void OnQueryPageSettings(object obj,
```

```
                                    QueryPageSettingsEventArgs gpsea)
        {
            if (gpsea.PageSettings.PrinterSettings.LandscapeAngle != 0)
                gpsea.PageSettings.Landscape ^= true;
        }

        void OnPrintPage(object obj, PrintPageEventArgs ppea)
        {
            Graphics grfx = ppea.Graphics;
            Font font = new Font("Times New Roman", 360);
            string str = iPageNNumber.ToString();
            SizeF sizef = grfx.MeasureString(str, font);

            grfx.DrawString(str, font, Brushes.Black,
                (grfx.VisibleClipBounds.Width - sizef.Width)/2,
                (grfx.VisibleClipBounds.Height - sizef.Height)/2);

            ppea.HasMorePages = iPageNNumber < iNumberPages;
            iPageNNumber+= 1;
        }
    }
}
```

Метод **MenuFilePrintOnClick** выполняется, когда пользователь выбирает Print из меню File. Вначале создается новый объект **PrintDocument** и новый объект **PrinterSelectionDialog**. Конструктор в **PrinterSelectionDialog** заполняет выпадающий список установленными принтерами. После этого метод устанавливает свойство **PrinterName** диалогового окна равным текущему принтеру:

```
dlg.PrinterName = prndoc.PrinterSettings.PrinterName;
```

Он становится выбранным элементом в выпадающем списке.

Если пользователь закрывает диалоговое окно, нажимая OK, свойство **PrinterName** свойства **PrinterSettings** объекта **PrintDocument** устанавливается равным выбранному принтеру:

```
prndoc.PrinterSettings.PrinterName = dlg.PrinterName;
```

Метод тогда устанавливает свойство **PrinterResolution** свойства **DefaultPageSettings PrintDocument** в разрешение «draft» для всего документа.

Метод **MenuFilePrintOnClick** включает в себя установку свойства **DocumentName** объекта **PrintDocument,** установку обработчиков событий для **PrintPage** и

QueryPageSettings, инициализацию номера страницы, и вызов метода **Print** у **PrintDocument,** чтобы начать печатать.

Следующий код, выполняемый в программой - обработчик событий **OnQueryPageSettings** для первой страницы. Если принтер поддерживает ландшафтный режим, метод переключает **Landscape** свойство объекта **PageSettings** переданное как свойство **QueryPageSettingsEventArgs:**

```
gpsea.PageSettings.Landscape ^= true;
```

После окончания работы метода **OnQueryPageSettings** обработчик событий **OnPrintPage** вызывается для первой страницы. Обработчик отображает номер страницы, напечатанный крупным шрифтом по центру.

Если принтер настроен для печати в альбомной ориентации по умолчанию, то первая страница будет напечатана в альбомной ориентации, вторая в книжной, и третья в альбомной. Обратите внимание, что **PrintPage** не делает ничего особенного кроме использования текущего свойства **VisibleClipBounds** объекта **Graphics,** чтобы выровнять текст по центру. **VisibleClipBounds** отражает текущую ориентацию для принтера.

Размеры страницы

Чтобы правильно рисовать текст и графику на странице принтера, нужно знать некоторые подробности о размере области, в которой можно рисовать. Печатать можно только в пределах определенной области, определенной пользователем.

Обработчики событий **PrintPage** получают объект типа **PrintPageEventArgs**. Одно из свойств этого класса – объект типа **Rectangle** под названием **PageBounds. PageBounds** равен свойству **Bounds** класса **PageSettings,** и определяет размеры физической страницы, принимая во внимание вертикальную или ландшафтную ориентацию, в сотых частях дюйма.

Класс **PageSettings** также включает объект под названием **Margins,** который показывает установленные пользователем расстояния от четырех краев страницы в сотых частях дюйма. По умолчанию, все четыре расстояния первоначально равны 100.

Свойство **MarginBounds** объекта **PrintPageEventArgs** - объект **Rectangle,** основанный на свойстве **PageBounds** с учетом размеров полей.

Объект **Graphics,** который получают из **PrintPageEventArgs,** установлен, чтобы напечатать на **печатаемой области** страницы. Начало координат этого объекта **Graphics** есть верхний левый угол печатаемой области страницы. Начало координат соответсвует свойству **VisibleClipBounds** объекта **Graphics**.

Таким образом, нужен прямоугольник, описывающий область печати относительно полного размера страницы. Однако, в объектах **PrinterSettings, PageSettings, PrintDocument** и **PrintPageEventArgs** нет данных о распределении областей, недоступных для печати, по краям страницы.

Можно рассчитать размеры прямоугольника, описывающего площадь страницы с учетом заданных пользователем полей, относительно **VisibleClipBounds** (значит, рассчитанный таким образом прямоугольник может быть использован рисующими методами объекта **Graphics).**

Если объект **PrintPageEventArgs** назван **ppea,** а объект **Graphics - grfx,** выражение:

```
(ppea.PageBounds.Width - grfx.VisibleClipBounds.Width)/2;
```

даст приблизительную ширину области, недоступной для печати с левого края

страницы, а:

```
(ppea.PageBounds.Height - grfx.VisibleClipBounds.Height)/2;
```

- аналогичное значение для правого края страницы. Вычитание этой пары значений соответственно из **ppea.MarginBounds.Left и ppeaMarginBounds.Top** позволяет получить координаты точки, приблизительно соответствующей верхнему левому углу области печати страницы, рассчитанной с учетом ширины пользовательских полей.

Вот расчет размеров прямоугольника с учетом ширины пользовательских полей:

```
RectangleF rectf = new RectangleF(
                ppea.MarginBounds.Left -
                 (ppea.PageBounds.Width - grfx.VisibleClipBounds.Width)/2,
                ppea.MarginBounds.Top -
                 (ppea.PageBounds.Height - grfx.VisibleClipBounds.Height)/2,
                ppea.MarginBounds.Width,
                ppea.MarginBounds.Height);
```

Это приблизительный расчет, так как он основан на допущении о равенстве ширины недоступных для печати полей слева и справа, а также сверху и снизу.

Класс *PrintController*

Класс **PrintController** («контроллер печати») определяет четыре метода:

Методы *PrintController*

```
void OnStartPrint(PrintDocument prndoc, PrintEventArgs pea);
Graphics OnStartPage(PrintDocument prndoc, PrintPageEventArgs ppea);
void OnEndPage(PrintDocument prndoc, PrintPageEventArgs ppea);
void OnEndPrint(PrintDocument prndoc, PrintEventArgs pea);
```

Когда программа инициирует печать, вызывая метод **Print** объекта **PrintDocument,** этот объект реагирует срабатыванием четырех событий, определенных в классе **PrintDocument: BeginPrint, QueryPageSettings, PrintPage** и **EndPrint.** Но **PrintDocument** также вызывает четыре метода объекта **PrintController** из свойства **PrintController:** после срабатывания своего события **BeginPrint -** метод **OnStartPrint,** до и после срабатывания каждого события **PrintPage -** соответственно методы **OnStartPage** и **OnEndPage** и, наконец, после срабатывания своего события **OnEndPrint -** метод **OnEndPrint.**

Метод **OnStartPage** объекта **PrintController,** в частности, отвечает за получение объекта **Graphics,** который, в конечном счете, передается обработчику события **PrintPage.** В сущности, объект **Graphics** определяет место, куда метод **PrintPage** выводит графическую информацию. Обычно выводимая графика поступает на принтер, за что и отвечает объект **PrintController.** В отличие от **PrintController,** у объекта

PreviewPrintController другие функции: он создает объект **Graphics** на основе битовой карты, представляющей страницу принтера. Именно так реализован в Windows Forms предварительный просмотр печатаемого документа.

По умолчанию свойство **PrintController** объекта **PrintDocument** содержит объект типа **PrintControllerWithStatusDialog.** Имя этого объекта раскрывает еще одну функцию объекта «контроллер печати»: он отображает диалоговое окно с именем печатаемого документа и номером страницы, которая печатается в данный момент. Если это диалоговое окно не нужно, поместите в свойство **PrintController** объекта **PrintDocument** объект типа **StandardPrintController,** который делает то же самое, что и **PrintControllerWithStatusDialog,** но при этом не выводит диалоговое окно.

Если нужно отображать ход печати не в диалоговом окне, а как-то иначе, можно породить соответствующий класс: от **StandardPrintController.** Вот пример программы для управления печатью, отображающей состояние печати на панели строки состояния:

```
//
// StatusBarPrintController.cs
//
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

namespace PrintWithStatusBar
{
    class StatusBarPrintController : StandardPrintController
    {
        StatusBarPanel statpanel;
        int iPageNumber;
        string strSaveText;

        public StatusBarPrintController(StatusBarPanel sbp): base()
        {
            statpanel = sbp;
        }

        public override void OnStartPrint(PrintDocument prndoc,
                                          PrintEventArgs pea)
        {
            strSaveText = statpanel.Text;
            statpanel.Text = "Starting printing";
            iPageNumber = 1;
```

```
            base.OnStartPrint(prndoc, pea);
        }

        public override Graphics OnStartPage(PrintDocument prndoc,
                                             PrintPageEventArgs ppea)
        {
            statpanel.Text = "Printing page " + iPageNumber++;
            return base.OnStartPage(prndoc, ppea);
        }
        public override void OnEndPage(PrintDocument prndoc,
                                       PrintPageEventArgs ppea)
        {
            base.OnEndPage(prndoc, ppea);
        }

        public override void OnEndPrint(PrintDocument prndoc,
                                       PrintEventArgs pea)
        {
            statpanel.Text = strSaveText;
            base.OnEndPrint(prndoc, pea);
        }
    }
}
```

Этот класс переопределяет все четыре метода **StandardPrintController,** и обеспечивает вызов соответствующих методов из базового класса. Это гарантирует, что объект «контроллер печати» не прекратит выполнять свои обычные действия. Единственное усовершенствование, реализованное в этой версии программы,- постоянное обновление панели строки состояния Объект «панель» нужен для конструктора класса.

Вот версия PrintThreePages, которая вместо диалогового окна для выбора принтера создает строку состояния с одной панелью:

```
//
// StatusBarPrintController.cs
//
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

namespace PrintWithStatusBar
```

```
{
    class StatusBarPrintController : StandardPrintController
    {
        StatusBarPanel statpanel;
        int iPageNumber;
        string strSaveText;

        public StatusBarPrintController(StatusBarPanel sbp): base()
        {
            statpanel = sbp;
        }

        public override void OnStartPrint(PrintDocument prndoc,
                                          PrintEventArgs pea)
        {
            strSaveText = statpanel.Text;
            statpanel.Text = "Starting printing";
            iPageNumber = 1;
            base.OnStartPrint(prndoc, pea);
        }

        public override Graphics OnStartPage(PrintDocument prndoc,
                                             PrintPageEventArgs ppea)
        {
            statpanel.Text = "Printing page " + iPageNumber++;
            return base.OnStartPage(prndoc, ppea);
        }
        public override void OnEndPage(PrintDocument prndoc,
                                       PrintPageEventArgs ppea)
        {
            base.OnEndPage(prndoc, ppea);
        }

        public override void OnEndPrint(PrintDocument prndoc,
                                       PrintEventArgs pea)
        {
            statpanel.Text = strSaveText;
            base.OnEndPrint(prndoc, pea);
        }
```

```
    }
}
```

Определяя свойство **PrintDocument** в ответ на выбор команды меню, эта версия программы также устанавливает свойство **PrintController:**

```
prndoc.PrintController = new StatusBarPrintController(sbarpanel);
```

Чтобы успеть убедиться в правильной работе программы, в **OnPrintPage** включен вызов метода **Sleep** из класса **Thread.**

Приложение не может реагировать на действия пользователя, пока метод **Print** объекта **PrintDocument** не вернет управления. Если программа отображает статус принтера в строке состояния, следует реализовать печать в фоновом режиме, для чего нужен второй поток.

Предварительный просмотр (Класс PreviewPrintController)

Для предварительного просмотра ключевым является свойство **PrintController** объекта **PrintDocument,** где по умолчанию находится объект **PrintControllerWithStatusDialog.** Необходимо, занести в свойство **PrintController** объект типа **PreviewPrintController.**

```
PreviewPrintController ppc = new PreviewPrintController();
prndoc.PrintController = ppc;
```

У класса **PreviewPrintController** имеется единственное свойство:

Свойство *PreviewPrintController*

Тип	**Свойство**	**Доступ**
bool	*UseAntiAlias*	Чтение/запись

Остальные свойства объекта **PrintDocument** устанавливают, как при обычной подготовке к печати. Далее следует инициировать печать вызовом метода **Print** объекта **PrintDocument.**

Объект «контроллер печати» отвечает за получение объекта **Graphics,** передаваемого объектом **PrintDocument** обработчику события **PrintPage.** Вместо получения объекта **Graphics** для принтера **PreviewPrintController** создает битовую карту для каждой страницы и получает объект **Graphics** для рисования на этой битовой карте. Таким образом, обработчику события **PrintPage** на самом деле передается объект **Graphics.**

После выполнения метода **Print** можно получить доступ к битовым картам страниц. Единственный метод объекта **PreviewPrintController,** не унаследованный им от других объектов, возвращает массив объектов **PreviewPageInfo:**

Метод *PreviewPrintController*

```
PreviewPageInfo[] GetPreviewPageInfo();
```

У класса **PreviewPageInfo** два свойства:

Свойства *PreviewPageInfo*

Тип	Свойство	Доступ
Image	*Image*	Чтение
Size	*PhysicalSize*	Чтение

Значение свойства **Image,** выраженное в пикселах, соответствует размеру страницы принтера, выраженному в тех же единицах. Свойство **PhysicalSize** указывает эти же размеры, выраженные в сотых долях дюйма. Есть другой способ предварительного просмотра: создание объекта типа **PrintPreviewDialog.**

```
PrintPreviewDialog predlg = new PrintPreviewDialog();
```

Вот некоторые свойства, реализуемые объектом **PrintPreviewDialog:**

Свойства *PrintPreviewDialog* (выборочно)

Тип	Свойство	Доступ
PrintDocument	*Document*	Чтение/запись
PrintPreviewControl	*PrintPreviewControl*	Чтение
bool	*UseAntiAlias*	Чтение/запись
bool	*HelpButton*	Чтение/запись

Ключевое свойство, которое нужно установить, **- Document.** В него нужно занести тот же объект **PrintDocument,** который применялся для печати и настройки параметров страницы. **PrintPreviewControl** также определяется в пространстве имен **SystemWindowsForms** и предоставляет элементы управления. Они в конечном счете появляются на форме, которая выводит битовые карты с изображениями страниц.

Вот обычный код, выполняющий инициализацию и инициирующий предварительный просмотр:

```
predlg.Document = prndoc;
predlg.ShowDialog();
```

Всю работу выполняет метод **ShowDialog.** Он извлекает из своего свойства **Document** объект **PrintDocument**, заносит в свойство **PrintController** объект **PreviewPrintController,** вызывает метод **Print** объекта **PrintDocument,** после чего показывает форму, отображающую битовые карты страниц при помощи набора элементов управления. При предварительном просмотре документ также можно напечатать. В этом случае диалоговое окно предварительного просмотра просто использует тот же самый объект **PrintDocument** с теми же обработчиками событий. Чтобы это сработало, нужно установить свойство **DocumentName** объекта **PrintDocument** перед вызовом метода **ShowDialog.**

2. Собственные элементы управления

Элементы управления Windows Forms – многократно используемые компоненты, которые инкапсулируют функциональность пользовательского интерфейса и используются в клиентских приложениях Windows. Windows Forms не только предоставляют множество готовых для использования элементов управления, но и инфраструктуру для разработки собственных элементов управления пользователя. Можно объединить существующие элементы управления, расширенные существующие элементы управления или создать собственные пользовательские элементы управления.

Элемент управления **Windows Forms** представляет собой класс, производный прямо или косвенно от класса **System.Windows.Forms.UserControl**. Известно, по крайней мере, три возможных подхода к разработке новых элементов управления **Windows Forms**.

1) *Объединение существующих элементов управления для создания составного элемента управления.* Составные элементы управления инкапсулируют пользовательский интерфейс, который может быть использован повторно в качестве элемента управления. Визуальные конструкторы предлагают широкие возможности поддержки создания составных элементов управления. Для создания составного элемента управления наследуют его от **System.Windows.Forms.UserControl**. Базовый класс **UserControl** обеспечивает клавиатурную маршрутизацию для дочерних элементов управления и позволяет им работать в группе.
2) *Расширение существующего элемента управления для его настройки или добавления новых функций*. Можно настроить любой элемент управления Windows Forms при помощи создания от него производного элемента и переопределения или добавления свойств, методов и событий.
3) Разработка элемента управления, который не объединяет и не расширяет существующие элементы управления. В этом случае элемент управления пользователя должен быть производным от базового класса **Control**. Можно также добавлять свойства, методы и события переопределения базового класса.

Базовый класс Control для элементов управления **Windows Forms** предоставляет все необходимое для визуального отображения в клиентских приложениях **Windows. Control** обеспечивает обработку окон и обработку маршрутизации сообщений, а также обеспечивает события клавиатуры и мыши, как и многие другие события пользовательского интерфейса. Он предоставляет дополнительные возможности построения макета формы и имеет специальные свойства для визуального отображения. Кроме того, он обеспечивает безопасность, поддержку работы с потоками. Благодаря такому разнообразию инфраструктуры, которое предоставляет базовый класс, разработать свой собственный элемент управления **Windows Forms** относительно просто.

Элемент управления играет роль графической связи между пользователем и программой. Элемент управления может предоставлять или обрабатывать данные, принимать данные, введенные пользователем, реагировать на события, а также выполнять любые другие функции, связывающие пользователя и приложение. Поскольку элемент управления по существу является компонентом с графическим интерфейсом, он может обрабатывать любую функцию, которую выполняет компонент, а также обеспечивать взаимодействие с пользователем.

Чтобы разработать элемент управления, выполняют следующие действия:

1. Определяют действия, которые должен выполнять элемент управления, или роль, которую он будет играть в приложении. Необходимо рассмотреть следующие факторы:
    - Какой вид должен иметь графический интерфейс?
    - Какие особые виды взаимодействия с пользователем будет обрабатывать элемент управления?
    - Существуют ли элементы управления, выполняющие нужные функции?
    - Можно ли получить нужные функции путем объединения нескольких элементов управления **Windows Forms**?
2. Если для элемента управления необходима объектная модель, определяют способ распределения функций в этой модели и разделяют их между элементом управления и вложенными объектами. Объектная модель может быть полезна при планировании сложного элемента управления или при необходимости объединения нескольких функций.
3. Определяют необходимый тип элемента управления (например, пользовательский элемент управления, нестандартный элемент управления, унаследованный элемент управления **Windows Forms**).
4. Представляют функции в качестве свойств, методов и событий элемента управления и его вложенных объектов или вспомогательных структур.
5. Если для элемента управления необходимо пользовательское оформление, то реализуют его.

Создание составного элемента управления

Составные элементы управления являются тем средством, с помощью которого можно создавать и многократно использовать пользовательские графические интерфейсы. Составной элемент управления - это компонент, имеющий визуальное представление. Он может состоять из одного или нескольких элементов управления форм **Windows Forms**, компонентов или блоков кода, позволяющих расширить функциональные возможности (например, проверять допустимость вводимых пользователем данных, изменять свойства отображения или выполнять другие действия, необходимые разработчику). Составные элементы управления можно поместить в формы **Windows Forms** точно так же, как и другие элементы управления.

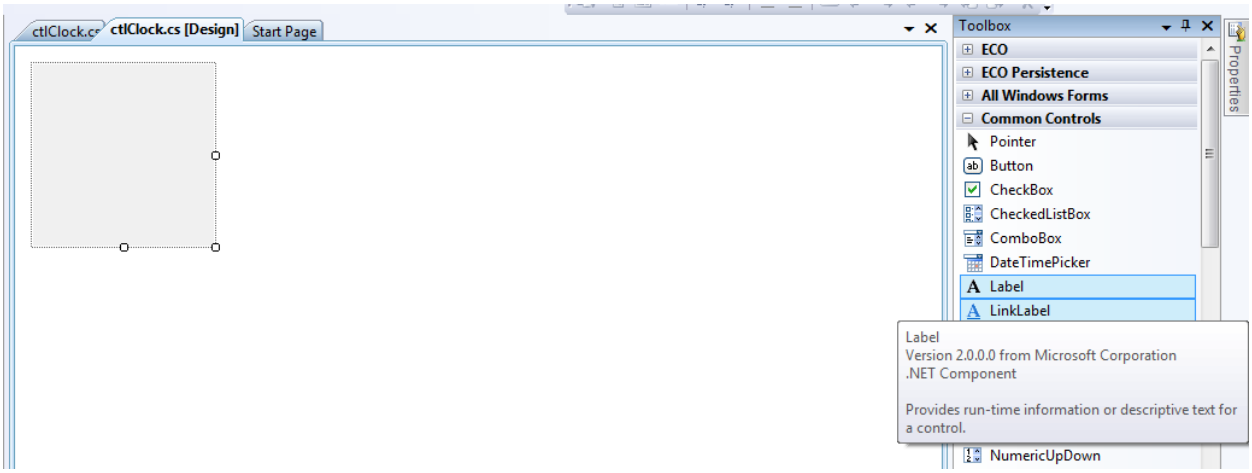
Разработка составных элементов управления предполагает объявление класса, производного от класса **UserControl** и использование Мастера **UserControl**, для добавления вложенных элементов управления с последующей настройкой образующих элементов.

Рассмотрим создание составного элемента управления на примера классов «Часы» и «Будильник». Создадим библиотеку элементов управления:

Создаем новый проект, из списка проектов Visual C# выбираем шаблон проекта Библиотека элементов управления Windows (**Windows Forms Control Library**) и даем проекту имя «ctlClockLib». В обозревателе решений щелкаем правой кнопкой мыши файл «UserControl1.cs» и переименовываем его на «ctlClock.cs».

Визуальный интерфейс - это важная часть составного элемента управления. Он создается путем добавления одного или нескольких элементов управления Windows на поверхность конструктора. Добавим элементы управления Label и Timer в составной элемент управления:

В обозревателе решений щелкаем правой кнопкой мыши элемент «ctlClock.cs» и выбираем пункт «**View Designer**». На панели элементов разворачиваем узел «Общие элементы управления» и помещаем на поверхность элемента надпись (**Label**).



Выбираем в конструкторе элемент label1. В окне "Свойства" задайте следующие значения свойств.

Свойство	**Значение**
Имя	lblDisplay
Текст	(пробел)
TextAlign	MiddleCenter
Размер шрифта	14

Выбираем компонент **timer1** и устанавливаем значение свойства **Interval** равным 1000, а значение свойства **Enabled** равным **true**. Свойство **Interval** управляет частотой тиканья компонента **Timer**. Каждый раз, когда компонент **timer1** тикает, выполняется код события **timer1_Tick**. Значение этого свойства - это количество миллисекунд между событиями **Tick**. Дважды щелкаем по компоненту **timer1**, чтобы перейти к событию **timer1_Tick.** Добавляем следующий код:

```
protected virtual void timer1_Tick(object sender, System.EventArgs e)
        {
            // Causes the label to display the current time.
            lblDisplay.Text = DateTime.Now.ToLongTimeString();
        }
```

Ключевое слово **virtual** указано, чтобы сделать метод доступным для переопределения

Созданный элемент управления «Часы» теперь инкапсулирует элемент управления **Label** и компонент **Timer**, каждый из которых имеет собственный набор наследуемых свойств. Хотя свойства этих элементов управления не будут доступны пользователям создаваемого элемента управления, можно создать и предоставить другим приложениям настраиваемые свойства, написав соответствующие блоки кода. Добавим в элемент управления свойства, позволяющие пользователю изменять цвет фона и текста.

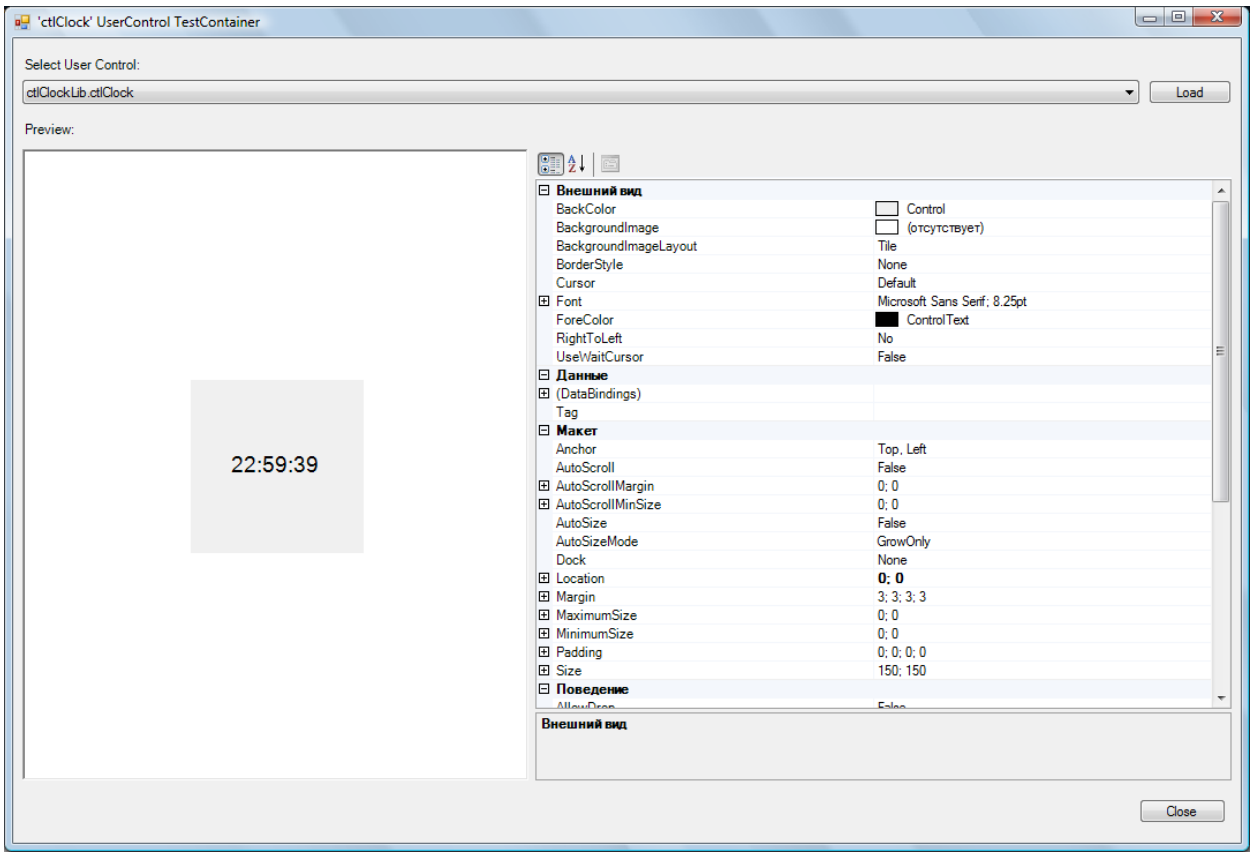
В коде элемента «ctlClock.cs» находим оператор **public partial class ctlClock**. Под открывающей скобкой вводим следующий код:

```
public partial class ctlClock : UserControl
    {
        private Color colFColor;
        private Color colBColor;

        public Color ClockBackColor
        {
            get
            {
                return colBColor;
            }

            set
            {
                colBColor = value;
                lblDisplay.BackColor = colBColor;
            }
        }

        public Color ClockForeColor
        {
            get
            {
                return colFColor;
            }
            set
            {
                colFColor = value;
                lblDisplay.ForeColor = colFColor;
            }
        }
```

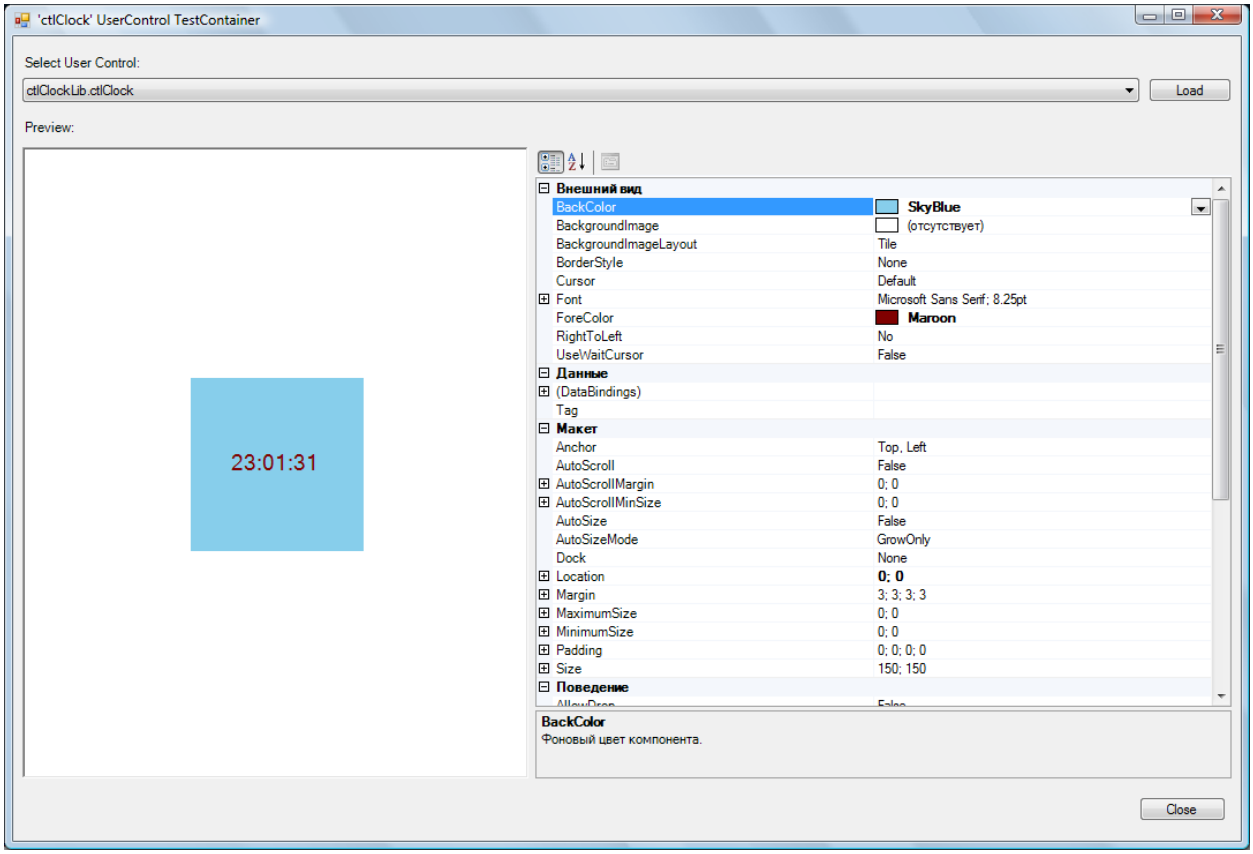
```
...
```

Эти операторы создают закрытые переменные. В них будут храниться значения создаваемых свойств. Приведенный выше код позволяет пользователям этого элемента управления получить доступ к настраиваемым свойствам **ClockForeColor** и **ClockBackColor**. Операторы **get** и **set** соответственно сохраняют и извлекают значение свойства, а также выполняют код, реализующий необходимую функцию.

Элементы управления не являются автономными приложениями; они должны быть включены в контейнер. Проверить поведение созданного элемента управления и его свойства можно с помощью тестового контейнера пользовательских элементов управления. Нажимаем клавишу F5 для построения проекта и запускаем элемент управления в тестовом контейнере пользовательских элементов управления.



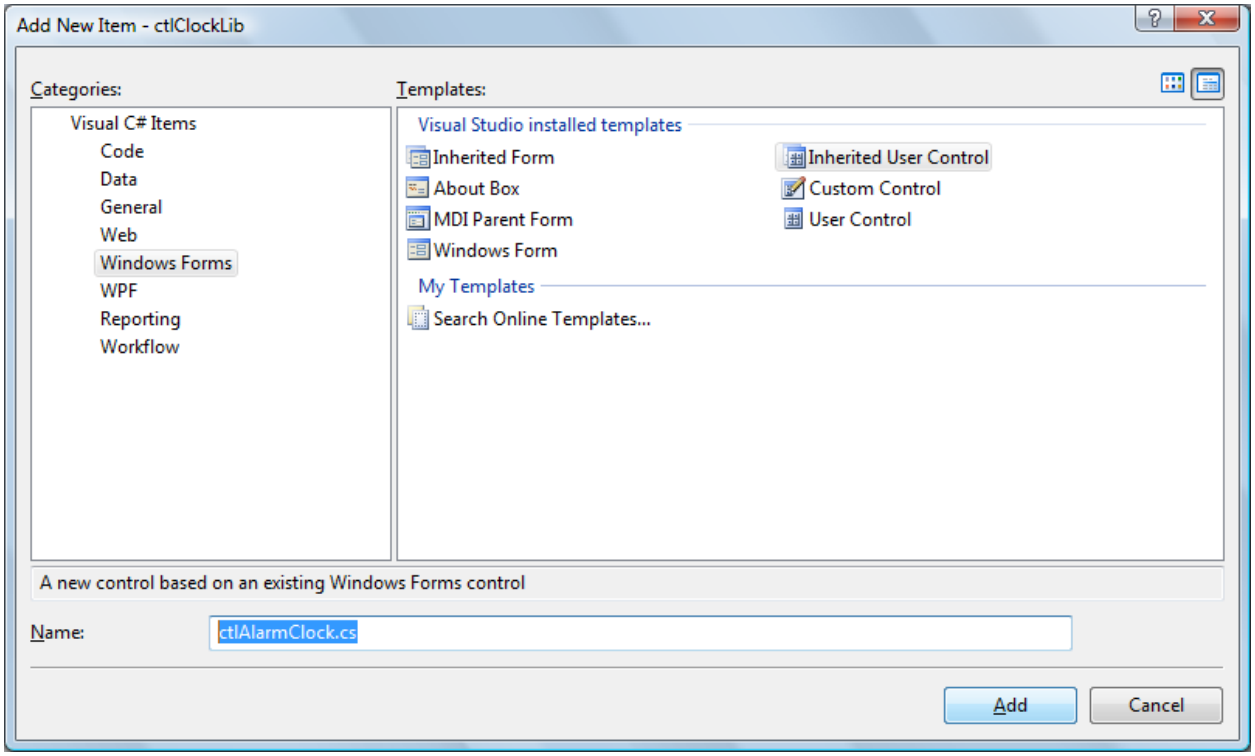
В таблице свойств тестового контейнера выбираем свойства **ClockBackColor** и **ClockForeColor**, чтобы вывести на экран цветовую палитру. Выбираем цвет с помощью щелчка мыши.



Рассмотрим, как на основе составного элемента управления построить другие элементы управления. Процесс создания производного класса на основе базового класса называется наследованием. Создадим составной элемент управления «Будильник» - **ctlAlarmClock**. Он будет производным от родительского элемента управления **ctlClock**. Расширим функциональные возможности элемента **ctlClock** путем переопределения методов родительского класса и добавления новых методов и свойств.

Первым шагом для создания наследуемого элемента управления является наследование от родительского элемента управления. При этом создается новый элемент управления, который не только имеет все свойства, методы и графические характеристики родительского элемента управления, но и служит основой для добавления или изменения функциональных возможностей.

Чтобы создать наследуемый элемент управления, добавим в проект пользовательский элемент управления, выбрав шаблон «Наследуемый пользовательский элемент управления».



В поле Имя вводим **ctlAlarmClock.cs** и нажимаем кнопку «Добавить». В диалоговом окне «Выбор компонентов для наследования» выбираем **ctlClock** под строкой Имя компонента.

Добавление свойств в наследуемый элемент управления выполняется точно так же, как и в случае с составным элементом управления. Теперь будет использован синтаксис объявления свойства для добавления двух свойств в элемент управления: **AlarmTime**, хранящее значение даты и времени срабатывания будильника, и **AlarmSet**, показывающее, установлен ли будильник.

В коде **ctlAlarmClock** находим оператор **public class**, и вводим следующий код:

```
public partial class ctlAlarmClock : ctlClockLib.ctlClock
    {
        private DateTime dteAlarmTime;
        private bool blnAlarmSet;
        private bool blnColorTicker;

        public DateTime AlarmTime
        {
            get
            {
                return dteAlarmTime;
            }
            set
            {
```

```
                dteAlarmTime = value;
            }
        }
        public bool AlarmSet
        {
            get
            {
                return blnAlarmSet;
            }
            set
            {
                blnAlarmSet = value;
            }
        }

        public ctlAlarmClock()
        {
            InitializeComponent();
        }
…
```

Наследуемый элемент управления имеет такой же визуальный интерфейс, как и его родительский элемент управления. Наследуемый элемент управления также наследует все элементы управления, содержащиеся в родительском элементе управления, но их свойства недоступны из наследуемого элемента, если они не были предоставлены явным образом. В графический интерфейс наследуемого составного элемента управления можно добавлять элементы так же, как и в графический интерфейс любого другого составного элемента управления. Далее в визуальный интерфейс будильника будет добавлен элемент управления надпись, который будет мигать при срабатывании будильника.

Открываем конструктор элемента управления **ctlAlarmClock**. Отметим, что хотя все свойства отображаются, они затенены. Это означает, что эти свойства являются собственными свойствами для **lblDisplay** и в окне свойств нельзя ни изменить их, ни обратиться к ним. Элементы управления, содержащиеся в составном элементе управления, по умолчанию являются закрытыми (**private**), и их свойства недоступны. Чтобы пользователи элемента управления имели доступ к его внутренним элементам управления, эти элементы управления должны быть объявлены как **public** или **protected**. Это позволит устанавливать и изменять свойства элементов управления, содержащихся в составном элементе управления, с помощью написания соответствующего кода.

С помощью мыши перетаскиваем элемент управления Label непосредственно под поле отображения. В окне "Свойства" задаем следующие значения свойств.

Свойство	**Значение**
Имя	lblAlarm
Текст	Wake up!!!
TextAlign	MiddleCenter
Видимость	false

Свойства и элемент управления, добавленные в предыдущих процедурах, позволяют реализовать функциональные возможности будильника в составном элементе управления. В этой процедуре в пользовательский элемент управления будет добавлен код, в котором текущее время сравнивается со временем срабатывания будильника. Если они совпадают, будильник начинает мигать. Переопределив метод **timer1_Tick** элемента управления **ctlClock** и добавив в него дополнительный код, можно расширить функциональные возможности наследуемого элемента управления **ctlAlarmClock,** сохранив в то же время все функциональные возможности элемента **ctlClock**.

Добавим свойство

```
private bool blnColorTicker;
```

Переопределим метод **timer1_Tick**:

```
protected override void timer1_Tick(object sender, System.EventArgs e)
        {
            base.timer1_Tick(sender, e);

            if (AlarmSet == false)
                return;
            else
            {
                if (AlarmTime.Date == DateTime.Now.Date &&
                    AlarmTime.Hour == DateTime.Now.Hour &&
                    AlarmTime.Minute == DateTime.Now.Minute)
                {
.
                    lblAlarm.Visible = true;
                    if (blnColorTicker == false)
                    {
                        lblAlarm.BackColor = Color.Red;
                        blnColorTicker = true;
                    }
                    else
```

```
                    {
                        lblAlarm.BackColor = Color.Blue;
                        blnColorTicker = false;
                    }
                }
                else
                {
                    lblAlarm.Visible = false;
                }
            }
        }
```

Этот код выполняет несколько задач. Оператор **override** указывает, что элемент управления должен использовать этот метод вместо метода, унаследованного от базового элемента управления. При вызове этого метода он вызывает переопределяемый им метод (с помощью вызова оператора **base.timer1_Tick**). Благодаря этому все функциональные возможности исходного элемента управления реализуются и в этом элементе. Затем выполняется дополнительный код, реализующий функциональные возможности будильника. При наступлении времени срабатывания будильника появляется мигающий элемент управления "Надпись".

Реализуем возможность отключения будильника. Добавим кнопку в элемент управления. Установим свойства кнопки следующим образом.

Свойство	**Значение**
Имя	btnAlarmOff
Текст	Turn off the alarm

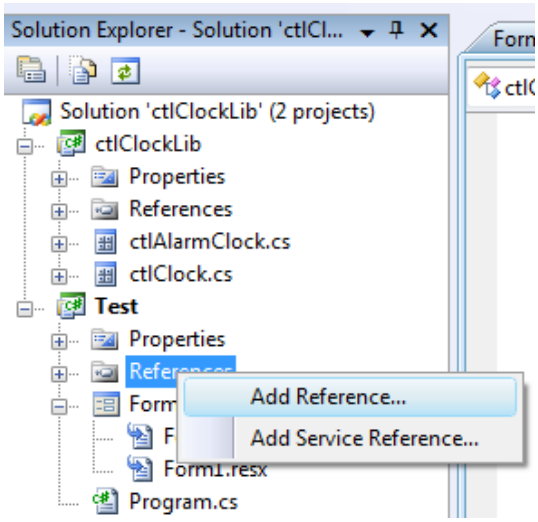
Определим метод **private void btnAlarmOff_Click**.

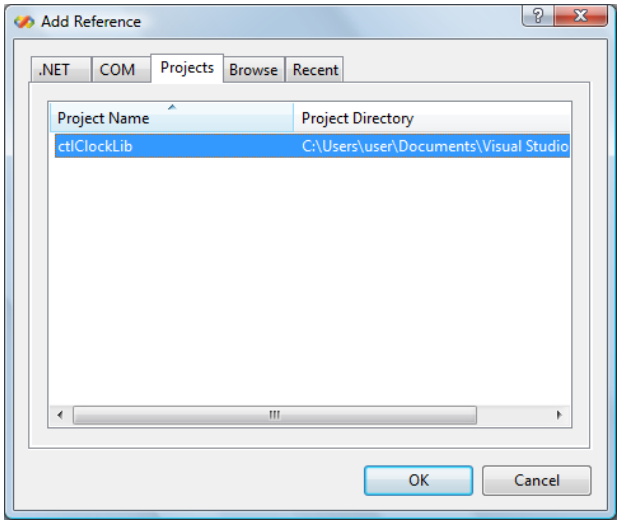
```
private void btnAlarmOff_Click(object sender, System.EventArgs e)
        {
            AlarmSet = false;
            lblAlarm.Visible = false;
        }
```

Для использования элемента управления необходимо включить его в форму. Как и стандартный составной элемент управления, наследуемый составной элемент управления не может быть автономным и должен быть включен в форму или другой контейнер. Поскольку элемент управления **ctlAlarmClock** имеет дополнительные функциональные возможности, для его тестирования требуется дополнительный код. Напишем простую программу для проверки функциональных возможностей элемента управления. Она будет устанавливать, и отображать значение свойства **AlarmTim**e элемента **ctlAlarmClock** и тестировать его функции.

Добавим в решение новый проект «Приложение Windows» и назовем его «**Test**».

В обозревателе решений добавим ссылку на тестовый проект **ctlClockLib**.





На панели элементов в разделе «Компоненты ctlClockLib» выбираем **ctlAlarmClock**, чтобы добавить в форму. Затем добавляем элемент **DateTimePicker** и элемент управления **Label**. Устанавливаем свойства этих элементов управления следующим образом:

Элемент управления	**Свойство**	**Значение**
label1	Текст	(пробел)
	Имя	lblTest
dateTimePicker1	Имя	dtpTest
	Формат	Time

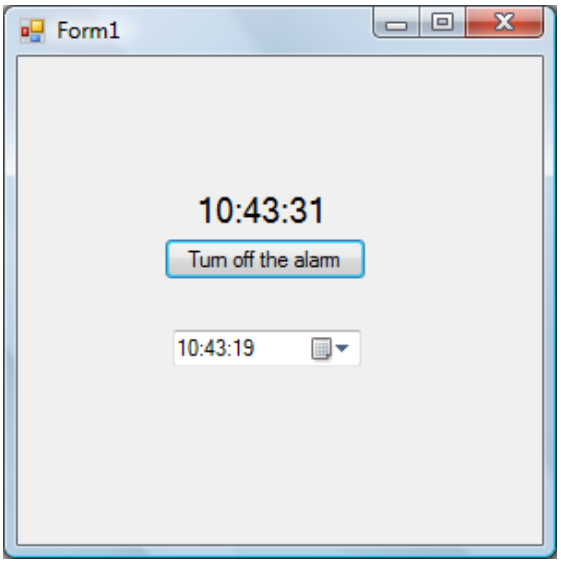
Определим метод **private void dtpTest_ValueChanged**.

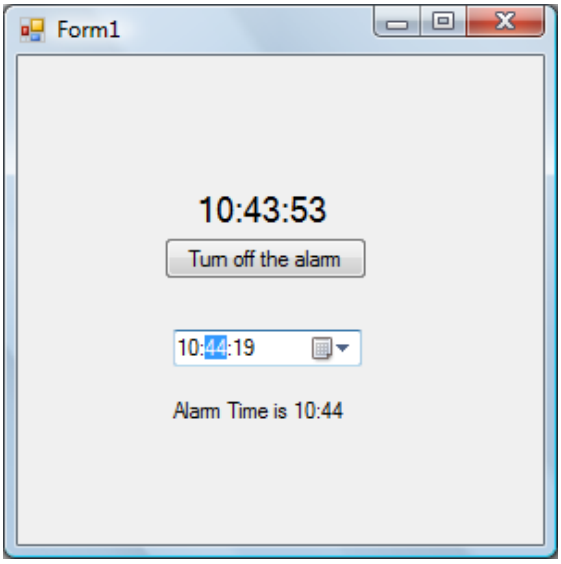
```
private void dtpTest_ValueChanged(object sender, System.EventArgs e)
        {
            ctlAlarmClock1.AlarmTime = dtpTest.Value;
            ctlAlarmClock1.AlarmSet = true;
            lblTest.Text = "Alarm Time is " +
                ctlAlarmClock1.AlarmTime.ToShortTimeString();
        }
```

В Обозревателе решений щелкаем правой кнопкой мыши **Test** и выберите «Назначить автозагружаемым проектом».

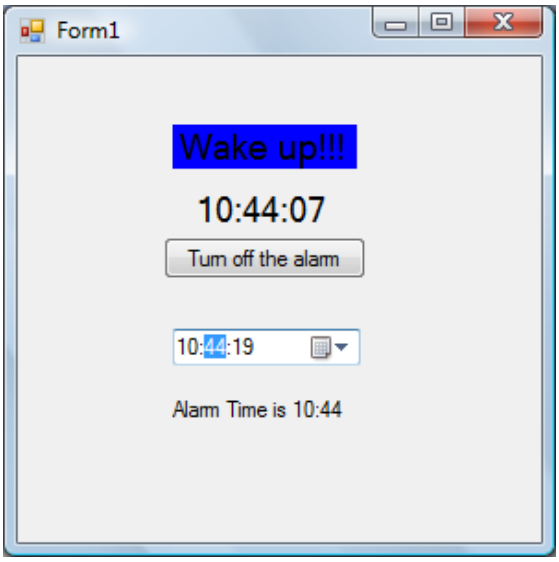
Запустим программу. В элементе управления **ctlAlarmClock** отображается текущее время, а в элементе управления **DateTimePicker** - начальное время. При помощи **DateTimePicker**, измените значение образом, чтобы оно было на одну минуту больше текущего времени, отображаемого элементом **ctlAlarmClock**. Когда текущее время станет

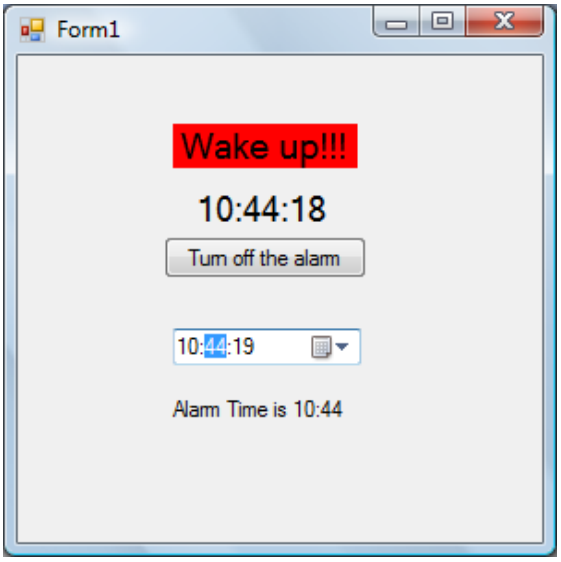
равным времени срабатывания будильника, **lblAlarm** начнет мигать. После этого можно отключить будильник, нажав **btnAlarmOff**.





До установки будильника *После установки будильника*





Будильник сработал

Расширение существующего элемента управления

Если нужно расширить функции существующего элемента управления, можно путем наследования создать элемент управления, производный от существующего. При наследовании существующего элемента управления наследуются все его функции и визуальные свойства. Например, если создан элемент управления, унаследованный из элемента управления Button, новый элемент управления будет выглядеть и действовать так же, как и стандартный элемент управления Button. Можно затем расширить или изменить функции нового элемента управления, реализовав собственные методы и свойства. В некоторых элементах управления можно также изменить внешний вид наследуемого элемента управления путем переопределения его метода **OnPaint**.

Рассмотрим пример создания элемента «Круглая кнопка». Чтобы создать наследуемый элемент управления создадим новый проект Windows «**RoundButton**». Проект имеет тип «Библиотека элементов управления Windows». В обозревателе решений щелкаем правой кнопкой мыши файл «**UserControl1.cs**» и переименовываем его на «**RoundButton.cs**».

В редакторе кода находим строку, которая задает класс **Control** в качестве базового класса для наследования. Изменяем имя базового класса на имя элемента управления, который нужно наследовать, в нашем случае это **Button**.

```
public partial class RoundButton: Button
```

Реализуем необходимые методы или свойства для элемента управления. Внешнее представление элемента управления меняется в зависимости от состояния элемента. Это состояние зависит от конкретных событий, происходящих с элементом управления. Внешний вид элемента управления определяется значением переменной indexB, которая меняет значение в рамках переопределяемых обработчиков событий. В нашем случае, кнопка будет менять цвет в зависимости от состояния.

```
public partial class RoundButton: Button
    {
        Brush[] clList;
        int indexB;

        public RoundButton(): base()
        {
            indexB = 0;
            clList = new Brush[] { Brushes.Red,
                Brushes.Pink, Brushes.Yellow, Brushes.Blue };
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            GraphicsPath gp = new GraphicsPath();
            gp.AddEllipse(10, 10, 50, 50);
            Region reg = new Region(gp);
            this.Region = reg;
            e.Graphics.FillRegion(clList[indexB],reg);
        }

        protected override void OnClick(EventArgs e)
        {
            indexB = 0;
            this.Refresh();
```

```
            base.OnClick(e);
        }

        protected override void OnMouseDown(MouseEventArgs e)
        {
            indexB = 1;
            this.Refresh();
            base.OnMouseDown(e);
        }

        protected override void OnGotFocus(EventArgs e)
        {
            indexB = 0;
            this.Refresh();
            base.OnGotFocus(e);
        }

        protected override void OnLostFocus(EventArgs e)
        {
            indexB = 0;
            this.Refresh();
            base.OnLostFocus(e);
        }

        protected override void OnMouseEnter(EventArgs e)
        {
            indexB = 2;
            this.Refresh();
            base.OnMouseEnter(e);
        }

        protected override void OnMouseLeave(EventArgs e)
        {
            indexB = 3;
            this.Refresh();
            base.OnMouseLeave(e);
        }
    }
```

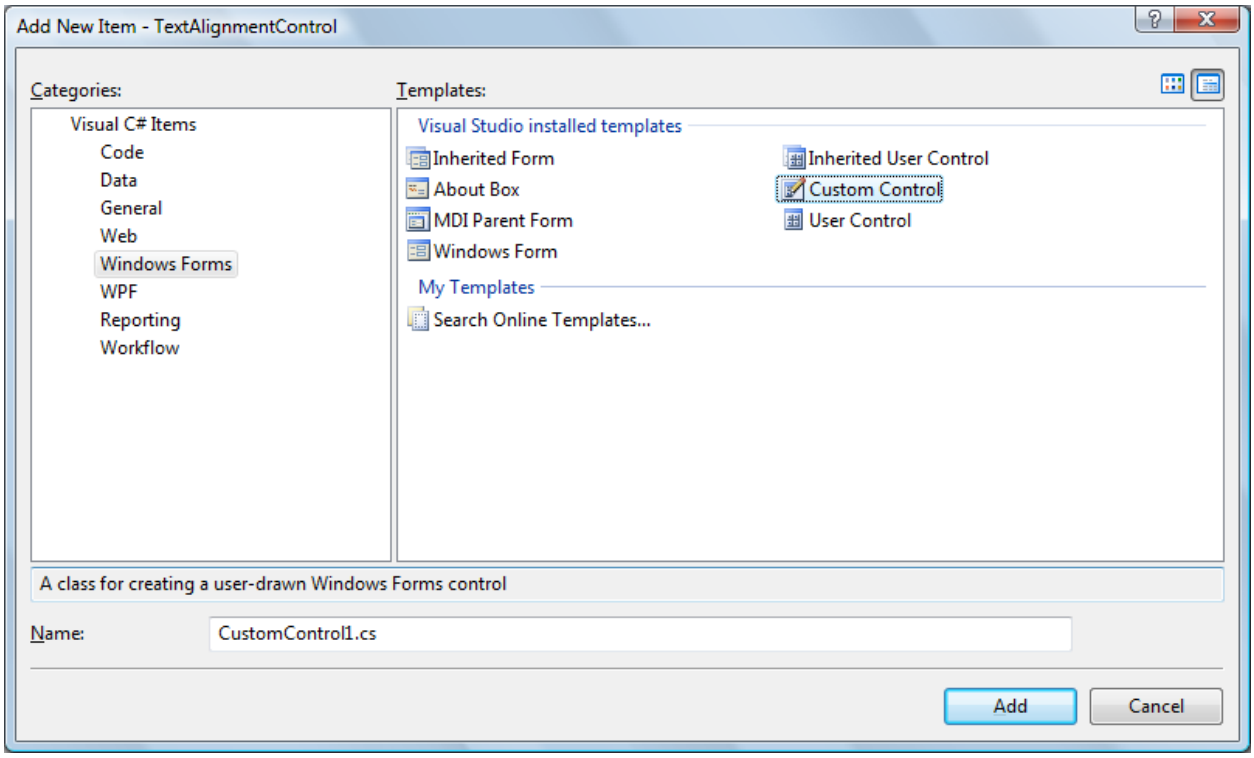
Так как внешний вид наследуемого элемента управления не схож с внешним видом родителя, то он изменен путем переопределения его метода **OnPaint**. Однако элемент все равно является кнопкой, наследуя все свойства родителя.

Разработка простого элемента управления

Если нужно полностью создать заново нестандартный элемент управления для использования в форме **Windows Forms**, следует наследовать класс **Control**. При наследовании класса **Control** требуется более тщательное планирование и реализация, однако при этом разработчику предлагается гораздо более широкий выбор вариантов. При наследовании класса **Control** наследуются самые простейшие функции, которые делают элементы управления работоспособными. Стандартные функции класса **Control** обрабатывают данные, вводимые пользователем посредством клавиатуры и мыши, определяют границы и размер элемента управления, предоставляют дескриптор окна, а также обеспечивают обработку и безопасность сообщений. В этот набор не входят функции оформления (фактического отображения графического интерфейса элемента управления) и специальные функции взаимодействия с пользователем. Разработчик должен обеспечить все эти возможности с помощью собственного кода.

Рассмотрим основные этапы разработки пользовательского элемента управления Windows Forms. Простейший элемент управления, разработанный в этом примере, позволяет изменять выравнивание своего свойства **Text**. Он не инициирует и не обрабатывает события.

Создаем новый проект «Библиотека элементов управления Windows». В меню Проект выбираем «Добавить класс». В диалоговом окне «Добавить новый элемент» выбираем «Настраиваемый элемент управления».



Определим дополнительные свойства. В следующем фрагменте кода определяется свойство с именем **TextAlignment** , которое используется **TextAlignmentControl** для форматирования значения свойства **Text**, унаследованного от **Control**.

```
private ContentAlignment alignmentValue = ContentAlignment.MiddleLeft;
```

Когда задается свойство, изменяющее отрисовку элемента управления, следует вызвать метод **Invalidate**, чтобы перерисовать элемент управления. Метод **Invalidate** определен в базовом классе **Control**.

Переопределяем защищенный метод **OnPaint**, наследуемый от **Control**, чтобы обеспечить логику отрисовки элемента управления. Если не переопределить метод **OnPaint**, элемент управления не сможет себя нарисовать. В следующем фрагменте кода метод **OnPaint** отображает значение свойства **Text**, унаследованного от **Control**, с выравниванием, определяемым значением поля **alignmentValue**.

```
protected override void OnPaint(PaintEventArgs e)
        {
            base.OnPaint(e);
            StringFormat style = new StringFormat();
            style.Alignment = StringAlignment.Near;
            switch (alignmentValue)
            {
                case ContentAlignment.MiddleLeft:
                    style.Alignment = StringAlignment.Near;
                    break;
                case ContentAlignment.MiddleRight:
                    style.Alignment = StringAlignment.Far;
                    break;
                case ContentAlignment.MiddleCenter:
                    style.Alignment = StringAlignment.Center;
                    break;
            }

            e.Graphics.DrawString(Text, Font,
                                  new SolidBrush(ForeColor),
                                  ClientRectangle, style);
        }
```

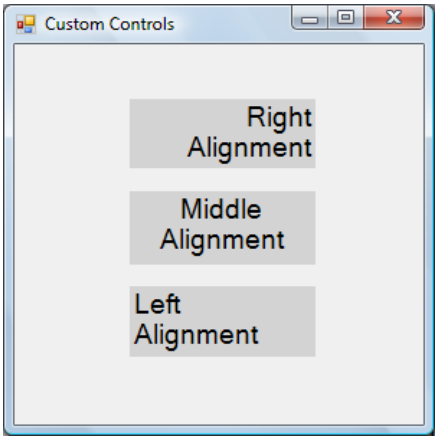
Задаем атрибуты для элемента управления. Атрибуты позволят визуальному конструктору отображать элемент управления и, соответственно, его свойства и события в режиме разработки. Следующий фрагмент кода применяет атрибуты для свойства **TextAlignment**. В конструкторе, таком как Visual Studio, атрибут **Category** позволяет свойству отображаться в соответствующей логической категории. Атрибут **Description** обеспечивает отображение строки описания внизу окна «Свойства», если выбрано свойство **TextAlignment**.

```
[
Category("Alignment"),
Description("Specifies the alignment of text.")
]
public ContentAlignment TextAlignment
{

    get
    {
        return alignmentValue;
    }
    set
    {
        alignmentValue = value;

        Invalidate();
    }
}
```

Протестируем разработанный элемент управления. Добавим в решение новый проект «Приложение Windows» и назовем его «**Test**». В обозревателе решений добавим ссылку на проект **TextAlignmentControl**. На панели элементов в разделе «Компоненты **TextAlignmentControl**» выбираем **TextAlignmentControl**, чтобы добавить в форму. Добавляем элемент три таких элемента управления. Устанавливаем свойства этих элементов управления следующим образом:

Элемент управления	**Свойство**	**Значение**
TextAlignmentControl1	Имя	RightAlignment
	Text	Right Alignment
	Text Alignment	MiddleRight
TextAlignmentControl2	Имя	MiddleAlignment
	Text	Middle Alignment
	Text Alignment	MiddleCenter
TextAlignmentControl3	Имя	LeftAlignment
	Text	Left Alignment
	Text Alignment	MiddleLeft

В Обозревателе решений щелкаем правой кнопкой мыши **Test** и выберите «Назначить автозагружаемым проектом». Запустим программу.



3. Домашнее задание

1) Реализовать текстовый редактор - клон Notepad, с возможностями печати, настройки параметров страницы и предварительного просмотра.
2) Разработать составной элемент управления аналогичный полю ввода калькулятора. Присутствует текстовое поле, кнопки с цифрами, десятичной точкой, и кнопка «Сброс». Вводимое значение проверяется на корректность и помещается в свойство элемента.
3) Модернизировать элемент управления «Круглая кнопка» добавив возможность вывода текста кнопки на ее поверхность, а так же возможность изменения размеров кнопки.
4) Расширить пример разработанного простого элемента управления, добавив ему возможность инициировать и обрабатывать события. К примеру, по щелчку изменяется выравнивание текста.

4. Экзаменационное задание

Разработать приложение, выполняющее посторенние блок-схем программы ,написанной на языке C++. Программа выполняет конвертацию из кода программы в блок-схему и, (в качестве дополнительной функциональности) редактирование блок-схем программ.

1. На вход программы приходит файл с расширением «.cpp».
2. Программа анализирует входной текстовый файл, разделяя его на функции.
3. Каждой функции соответствует своя построенная блок-схема.
4. Комментарии в файле кода игнорируются.
5. При анализе текста функции, код разбивается на соответствующие блоки, с целью последующей отрисовки.
6. Текстовое содержимое блоков масштабируется в соответствии их размерам.
7. Построенная блок-схема выводится на экран.
8. Присутствует возможность сохранить блок-схему в файл.

Рекомендации: при решении задачи разработки программы спроектировать иерархию классов для хранения структур - блоков кода. Классы этой иерархии должны обладать возможностью визуализации на форме. Возможен вариант, когда классы иерархии являются пользовательскими элементами управления.

Программа должна обладать следующей базовой функциональностью:

1. Чтение кода из текстового файла
2. Построение блок-схемы
3. Вывод блок-схемы на экран
4. Сохранение построенной блок-схемы в файл

Программа может обладать следующей дополнительной функциональностью:

1. Возможность построения блок-схемы вручную
2. Возможность редактирование автоматически построенной блок-схемы
3. Возможность редактирования кода внутри блоков схемы
4. Печать построенной блок-схемы

Примечание: считается, что текстовый файл на входе программы содержит **корректную** программу на языке C++, состоящую из **функций**. Код не содержит определений классов. Корректность полученного кода не проверяется.

Основные элементы схем алгоритма

Наименование	Обозначение	Функция
Терминатор (пуск-останов)		Элемент отображает вход из внешней среды или выход из нее (наиболее частое применение − начало и конец программы). Внутри фигуры записывается соответствующее действие.
Процесс		Выполнение одной или нескольких операций, обработка данных любого вида (изменение значения данных, формы представления, расположения). Внутри фигуры записывают непосредственно сами операции.
Решение		Отображает решение или функцию переключательного типа с одним входом и двумя или более альтернативными выходами, из которых только один может быть выбран после вычисления условий, определенных внутри этого элемента. Вход в элемент обозначается линией, входящей обычно в верхнюю вершину элемента. Выходов два или три, каждый выход обозначается линией, выходящей из оставшихся вершин (боковых и нижней).
Предопределенный процесс		Символ отображает выполнение процесса, состоящего из одной или нескольких операций, который определен в другом месте программы (в подпрограмме, модуле). Внутри символа записывается название процесса и передаваемые в него данные. Например − вызов процедуры или функции.
Данные (ввод-вывод)		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод). Данный символ не определяет носителя данных (для указания типа носителя данных используются специфические символы).

Блок-схемы обычно состоят из следующих блоков:

1. Начало алгоритма – Овал со словом «Begin»
2. Конец алгоритма – Овал со словом «End»
3. Ввод данных – параллелограмм с вводимыми данными

4. Блок вывода данных – Такой же, как и блок ввода, либо прямоугольник со скругленными углами с выводимыми данными.
5. Блок кода – прямоугольник с кодом
6. Блок ветвления – ромб с проверяемым условием
7. Блок вызова функции – прямоугольник с полями по бокам с именем вызываемой функции и параметрами.

Блок начала алгоритма имеет один выход и ни одного входа. Блок конца алгоритма имеет один вход и ни одного выхода. Все остальные блоки, кроме блока ветвления, имеют один вход и один выход. Блок ветвления имеет один вход, и два выхода, соответствующих истинному и ложному значению проверяемого условия.

Пример

Код на входе:

```
T BinarySearchR(T arr[], T searchKey, int low,int high)
{
	int middle;
	if (low <= high)
	{
		middle = (low + high) / 2;

		if (searchKey == arr[middle])
			return middle;
		else
		{
			if (searchKey < arr[middle])
				high = middle - 1;
			else
				low = middle + 1;

			return BinarySearchR(arr, searchKey, low, high);
		}
	}
	return -1;
}
```

Блок-схема на выходе: