# Программирование на языке Delphi



Решаемая на компьютере задача реализуется в виде прикладной программы, которую для краткости называют приложением. В основе разработки приложения в среде Delphi лежит проект. Центральной частью проекта является форма, на которую помещаются необходимые для решения конкретной задачи компоненты. В такой последовательности — проект - формы - компоненты — мы и рассмотрим процесс создания приложения в среде Delphi. По ходу изложения материала мы будем часто обращаться к примеру с вычислением идеального веса, который был рассмотрен в первой главе. Если вы его забыли, перечитайте первую главу еще раз.

## 7.1. Проект

## 7.1.1. Понятие проекта

Приложение собирается из многих элементов: форм, программных модулей, внешних библиотек, картинок, пиктограмм и др. Каждый элемент размещается в отдельном файле и имеет строго определенное назначение. Набор всех файлов, необходимых для создания приложения, называется *проектом*. Компилятор последовательно обрабатывает файлы проекта и строит из них выполняемый файл. Основные файлы проекта можно разделить на несколько типов:

- *Файлы описания форм* — текстовые файлы с расширением DFM, описывающие формы с компонентами. В этих файлах запоминаются начальные значения свойств, установленные вами в окне свойств.
- *Файлы программных модулей* — текстовые файлы с расширением PAS, содержащие исходные программные коды на языке Delphi. В этих файлах вы пишите методы обработки событий, генерируемых формами и компонентами.
- *Главный файл проекта* — текстовый файл с расширением DPR, содержащий главный программный блок. Файл проекта подключает все используемые программные модули и содержит операторы для запуска приложения. Этот файл среда Delphi создает и контролирует сама.

На основании сказанного можно изобразить процесс создания приложения в среде Delphi от постановки задачи до получения готового выполняемого файла (рисунок 7.1):

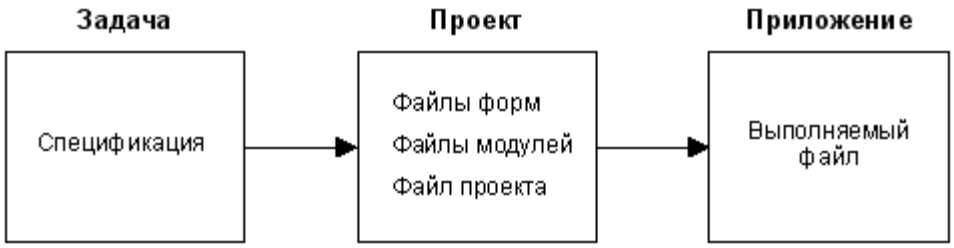


*Рисунок 7.1. Процесс создания приложения в среде Delphi*

Давайте рассмотрим назначение и внутреннее устройство файлов проекта. Это поможет вам легче ориентироваться в проекте.

## 7.1.2. Файлы описания форм

Помните, с чего вы начинали знакомство со средой Delphi? Конечно, с формы. Итак, первая составная часть проекта — это текстовый файл с расширением DFM, описывающий форму. В DFM-файле сохраняются значения свойств формы и ее компонентов, установленные вами в окне свойств во время проектирования приложения. Количество DFM-файлов равно количеству используемых в приложении форм. Например, в нашем примере об идеальном весе используется только одна форма, поэтому и DFM-файл только один — Unit1.DFM.

Если вы желаете взглянуть на содержимое DFM-файла, вызовите у формы контекстное меню щелчком правой кнопки мыши и выберите команду **View as Text** (рисунок 7.2).

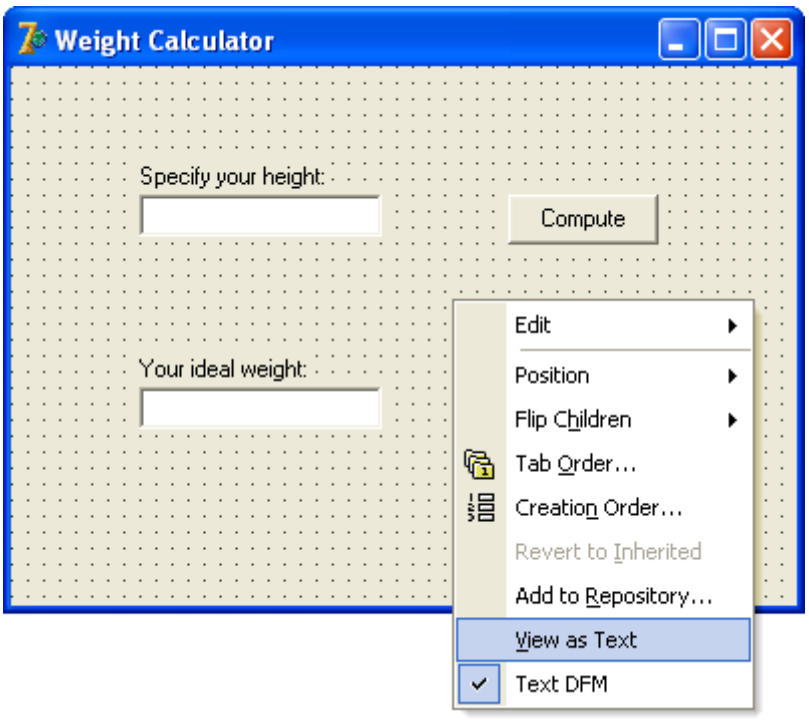


*Рисунок 7.2. Переход к текстовому представлению формы с помощью команды View as Text контекстного меню*

В ответ среда Delphi вместо графического изображения формы покажет следующий текст в редакторе кода**:**

```
object Form1: TForm1
  Left = 250
  Top = 150
  Width = 400
  Height = 303
  Caption = 'Weight Calculator'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Label1: TLabel
    Left = 64
    Top = 48
    Width = 93
    Height = 13
    Caption = 'Specify your height:'
  end
  object Label2: TLabel
    Left = 64
    Top = 144
    Width = 84
    Height = 13
    Caption = 'Your ideal weight:'
  end
  object Button1: TButton
    Left = 248
    Top = 64
    Width = 75
```

```
    Height = 25
    Caption = 'Compute'
    TabOrder = 0
    OnClick = Button1Click
  end
  object Button2: TButton
    Left = 248
    Top = 160
    Width = 75
    Height = 25
    Caption = 'Close'
    TabOrder = 1
  end
  object Edit1: TEdit
    Left = 64
    Top = 64
    Width = 121
    Height = 21
    TabOrder = 2
  end
  object Edit2: TEdit
    Left = 64
    Top = 160
    Width = 121
    Height = 21
    TabOrder = 3
  end
end
```

Несмотря на столь длинный текст описания, разобраться в нем совсем не сложно. Здесь на специальном языке задаются исходные значения для свойств формы Form1 и ее компонентов Button1, Button2, Edit1, Edit2, Label1, Label2. Большего знать не требуется, поскольку вы всегда будете использовать визуальные средства проектирования и работать с графическим представлением формы, а не с текстовым описанием. Раз так, давайте поспешим вернуться к графическому представлению, не внося в текст никаких изменений. Для этого вызовите контекстное меню редактора кода и выберите команду **View as Form** (рисунок 7.3).

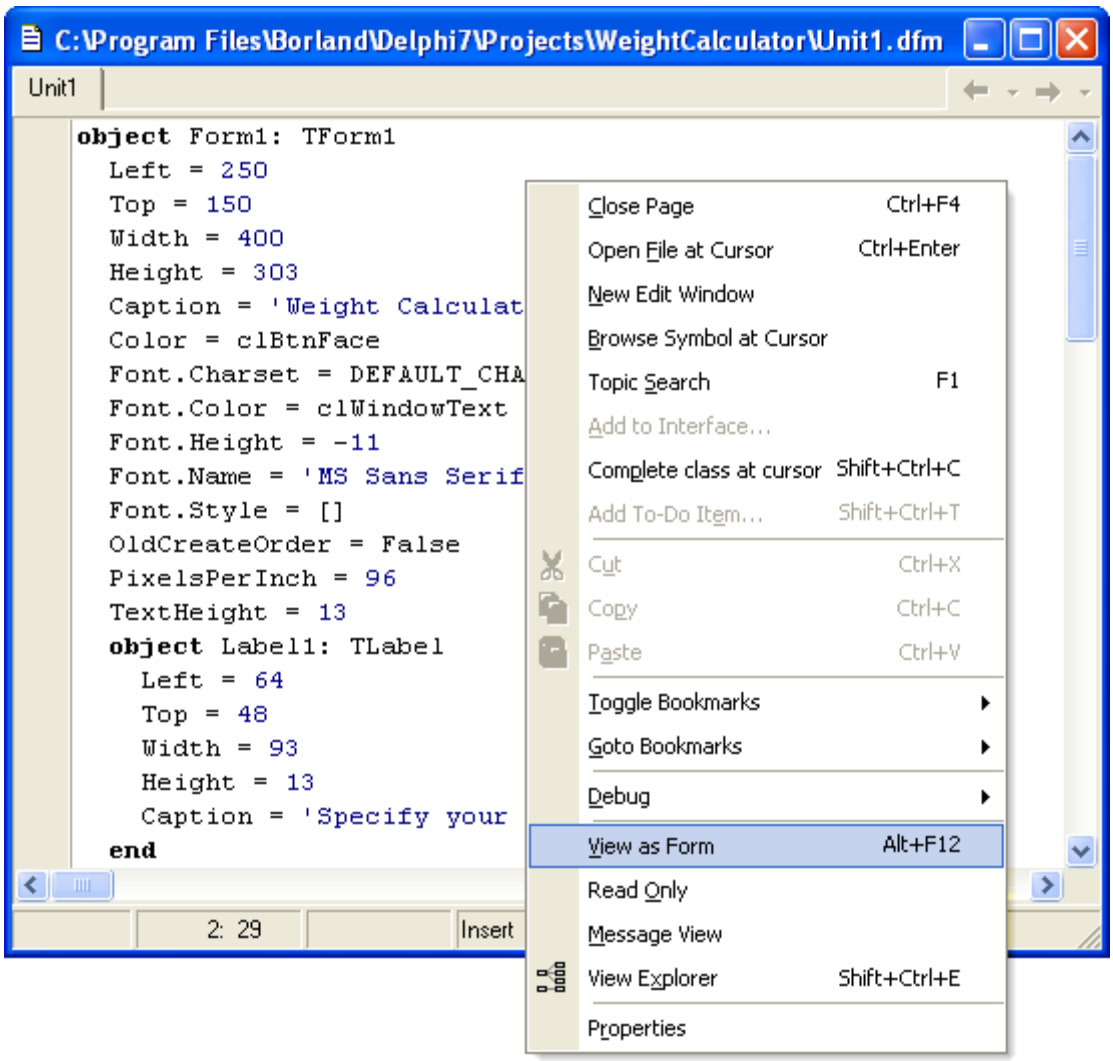


*Рисунок 7.3. Переход к графическому представлению формы с помощью команды View as Form контекстного меню*

На экране снова появится графический образ формы. Если вы все-таки внесли корректировки в текст, то они отразятся на внешнем виде формы.

Файл описания формы (DFM-файл) нужен только на этапе проектирования. При сборке приложения описание формы из DFM-файла помещается в специальную область данных выполняемого файла (область ресурсов). Когда во время работы приложения происходит создание формы, ее описание извлекается из области ресурсов и используется для инициализации формы и ее компонентов. В результате форма отображается на экране так, как вы задали при проектировании.

### 7.1.3. Файлы программных модулей

Каждой форме в проекте соответствует свой *программный модуль* (unit), содержащий все относящиеся к форме объявления и методы обработки событий, написанные на языке Delphi. Программные модули размещаются в отдельных файлах с расширением PAS. Их количество может превышать количество форм. Почему? Потому, что в ряде случаев программные модули могут и не относиться к формам, а содержать вспомогательные процедуры, функции, классы и проч. Наша задача об идеальном весе очень простая, поэтому в ней имеется только один программный модуль, связанный с формой. Не поленитесь изучить его внимательно:

```
unit Unit1;

interface
```

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  // Алгоритм вычисление идеального веса
  Edit2.Text := IntToStr(StrToInt(Edit1.Text) - 100 - 10);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

end.
```

Дадим необходимые комментарии к тексту программного модуля. В самом начале после ключевого слова **unit** записывается имя модуля

```
unit Unit1;
```

Ни в коем случае не изменяйте это имя вручную. Среда Delphi требует, чтобы имя модуля совпадало с именем файла, поэтому если вы хотите переименовать модуль, сохраните его в файле с новым именем, воспользовавшись командой меню **File | Save As...**. Среда Delphi сама подставит после слова **unit** новое имя. После этого удалите старый модуль.

Содержание интерфейсной секции модуля (**interface**) начинается с подключения стандартных модулей библиотеки VCL, в которых определены часто вызываемые подпрограммы и классы помещенных на форму компонентов.

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
```

Среда Delphi формирует список модулей без вашего участия и автоматически пополняет его, когда вы добавляете на форму новые компоненты. Тем не менее, список подключенных модулей можно изменять прямо в редакторе кода (вручную).

Смотрим дальше. В разделе описания типов (**type**) объявлен класс формы. По умолчанию он называется **TForm1** и порожден от стандартного класса **TForm**.

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Помещенные на форму компоненты представлены полями формы. У нас на форме шесть компонентов, поэтому и полей в описании класса тоже шесть. Имена полей совпадают с именами компонентов, заданными в окне свойств.

После полей следуют заголовки методов обработки событий. Название каждого такого метода среда Delphi формирует автоматически на основании имени компонента и имени генерируемого им события. Например, для кнопки **Button1** метод обработки события **OnClick** называется **Button1Click**.

Обратите внимание, что поля, представляющие компоненты формы, а также методы обработки событий получают атрибут видимости **published** (он принимается по умолчанию для всех наследников **TForm**). Благодаря этому вы можете работать с ними на визуальном уровне, например, видеть их имена в окне свойств. Поскольку среда Delphi сама управляет содержимым секции **published**, никогда не модифицируйте эту секцию вручную (в редакторе кода), пользуйтесь визуальными инструментами: палитрой компонентов и окном свойств. Запомните:

- когда вы помещаете на форму компоненты, среда Delphi сама добавляет соответствующие поля в описание класса формы, а когда вы удаляете компоненты с формы, среда удаляет поля из описания класса;
- когда вы определяете в форме или компонентах обработчики событий, среда Delphi сама определяет в классе соответствующие методы, а когда вы удаляете весь код из методов обработки событий, среда Delphi удаляет и сами методы.

Для вашего удобства в классе формы заранее объявлены пустые секции **private** и **public**, в которых вы можете размещать любые вспомогательные поля, методы и свойства. Среда Delphi их "в упор не видит", поэтому с ними можно работать только на уровне программного кода. Вы можете помещать в секцию **private** атрибуты, которые нужны только самой форме, а в секцию **public** — атрибуты, которые нужны еще и другим формам и модулям.

После описания класса идет объявление собственно объекта формы:

```
var
  Form1: TForm1;
```

Переменная **Form1** — это ссылка на объект класса **TForm1**, конструирование которого выполняется в главном файле проекта – DPR-файле (см. далее).

На этом содержание интерфейсной секции модуля заканчивается и начинается раздел реализации (**implementation**). Сначала в нем подключается файл описания формы:

```
{$R *.dfm}
```

Пожалуйста, не подумайте, что эта директива подключает все файлы с расширением DFM. Подключается лишь один DFM-файл, в котором описана форма данного модуля. Имя DFM-файла получается заменой звездочки на имя модуля, в котором записана директива.

Далее следует реализация методов обработки событий. Пустые заготовки для них среда Delphi создает сама одновременно с добавлением заголовков в класс формы. Вы же наполняете их содержанием.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // Алгоритм вычисление идеального веса
  Edit2.Text := IntToStr(StrToInt(Edit1.Text) - 100 - 10);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

Внимание! Если вы хотите удалить метод обработки события и убрать ссылки на него, просто сделайте метод пустым — удалите весь написанный вами код, включая комментарии, и объявления локальных переменных. При сохранении или компиляции проекта среда Delphi сама выбросит из текста пустые методы.

При внимательном изучении исходного текста модуля остается невыясненным один вопрос: как обеспечивается вызов методов **Button1Click** и **Button2Click** при нажатии на форме кнопок, ведь в тексте модуля отсутствует даже намек на это. Все очень просто. Загляните в DFM-файл. Кроме установки значений свойств вы найдете установку и обработчиков событий.

```
object Button1: TButton
  ...
  OnClick = Button1Click
end
object Button2: TButton
  ...
  OnClick = Button2Click
end
```

Благодаря этому описанию обеспечивается привязка методов формы к соответствующим событиям. Кстати, смысл приведенного фрагмента описания становится более прозрачным, если вспомнить, что события в языке Delphi — это на самом деле свойства, но их значениями являются указатели на методы. Таким образом, установка событий мало чем отличается от установки свойств формы, ведь по природе это одно и то же.

Мы достаточно глубоко погрузились во внутреннее устройство файлов описания форм и файлов программных модулей и, признаемся, сделали это намеренно, чтобы дать вам полное понимание вопроса, не заставляя принимать на веру далеко неочевидные вещи. А сейчас пора подняться на уровень проекта и посмотреть, что же объединяет все эти файлы.

### 7.1.4. Главный файл проекта

Для того чтобы компилятор знал, какие конкретно файлы входят в проект, необходимо какое-то организующее начало. И оно действительно есть. Это так называемый *файл проекта*, имеющий расширение DPR (сокр. от Delphi Project). Он представляет собой главный программный файл на языке Delphi, который подключает с помощью оператора **uses** все файлы модулей, входящих в проект. Для каждого проекта существует только один DPR-файл.

Когда вы по команде **File | New | Application** начинаете разработку нового приложения, среда Delphi автоматически создает файл проекта. По мере создания новых форм содержимое этого файла видоизменяется автоматически. Когда вы закончите работу и будете готовы компилировать проект, в DPR-файле будет находиться перечень программных модулей, которые будут поданы на вход компилятору. Чтобы увидеть содержимое DPR-файла нашего приложения,

вычисляющего идеальный вес, выберите в меню среды Delphi команду **Project | View Source**. В редакторе кода появится новая страница со следующим текстом:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Немного прокомментируем этот текст. Подключение модуля **Forms** обязательно для всех программ, так как в нем содержится определение объекта **Application**. Этот объект лежит в основе любого графического приложения и доступен на протяжении всей его работы.

Подключаемый следом модуль Unit1 содержит определение формы. Название формы приводится в фигурных скобках. Директива **in** указывает на то, что модуль является необходимой частью проекта и существует в виде исходного текста на языке Delphi.

Директива {$R *.res} подключает к результирующему выполняемому файлу так называемые ресурсы, в данном случае значок приложения. Этот значок будет виден на Панели Задач.

Дальше следует главный программный блок, содержащий вызовы трех методов объекта **Application**. Вызов метода **Initialize** подготавливает приложение к работе, метод **CreateForm** загружает и инициализирует форму **Form1**, а метод **Run** активизирует форму и начинает выполнение приложения. Фактически время работы метода **Run** — это время работы приложения. Выход из метода **Run** происходит тогда, когда пользователь закрывает главную форму приложения; в результате приложение завершается.

Внимание! Никогда не изменяйте DPR-файл вручную. Оставьте эту работу для среды Delphi. Добавление и удаление модулей, а также управление созданием форм осуществляется с помощью команд и диалоговых окон среды.

### 7.1.5. Другие файлы проекта

Выше мы рассмотрели основные файлы проекта. Кроме них существует ряд дополнительных файлов:

- Файл с расширением DOF (сокр. от Delphi Options File), где хранятся заданные программистом параметры компиляции и сборки проекта;
- Файл с расширением DSK (сокр. от англ. Desktop), где хранятся настройки среды Delphi для данного проекта. Чтобы среда Delphi сохраняла свои настройки в DSK-файле, выберите в меню команду **Tools | Environment Options...** и в диалоговом окне **Environment Options** на вкладке **Preferences** в группе **Autosave options** отметьте пункт **Project Desktop**.
- Файл с расширением CFG (сокр. от англ. Configuration), где хранятся настройки для консольного варианта компилятора.
- Файл с расширением DCI (сокр. от англ. Delphi CodeInsight), где среда Delphi хранит сделанные вами настройки для программного "суфлера" (CodeInsight).
- Файл с расширением DCT (сокр. от англ. Delphi Component Templates), где хранятся ваши домашние заготовки компонентов.
- Файл с расширением DMT (сокр. от англ. Delphi Menu Templates), где хранятся ваши домашние заготовки меню.
- Файл с расширением DRO, где хранятся настройки и ваши добавки к хранилищу компонентов.

- Файл с расширением TODO — записная книжка для хранения заданий на программирование и коротких примечаний.
- Файл с расширением DDP (сокр. от англ. Delphi Diagram Portfolio), где хранятся графические схемы, наглядно поясняющие взаимосвязи между компонентами.
- Файл ресурсов с расширением RES (сокр. от RESource). В нем, например, хранится значок приложения, который отображается на Панели Задач. О том, как задать значок приложения, мы расскажем при обсуждении вопросов управления проектом.

В проект могут входить также логически автономные элементы: точечные рисунки (BMP-файлы), значки (ICO-файлы), файлы справки (HLP-файлы) и т.п., однако ими управляет сам программист.

Теперь можно уточнить рисунок, отражающий состав проекта (рисунок 7.4):

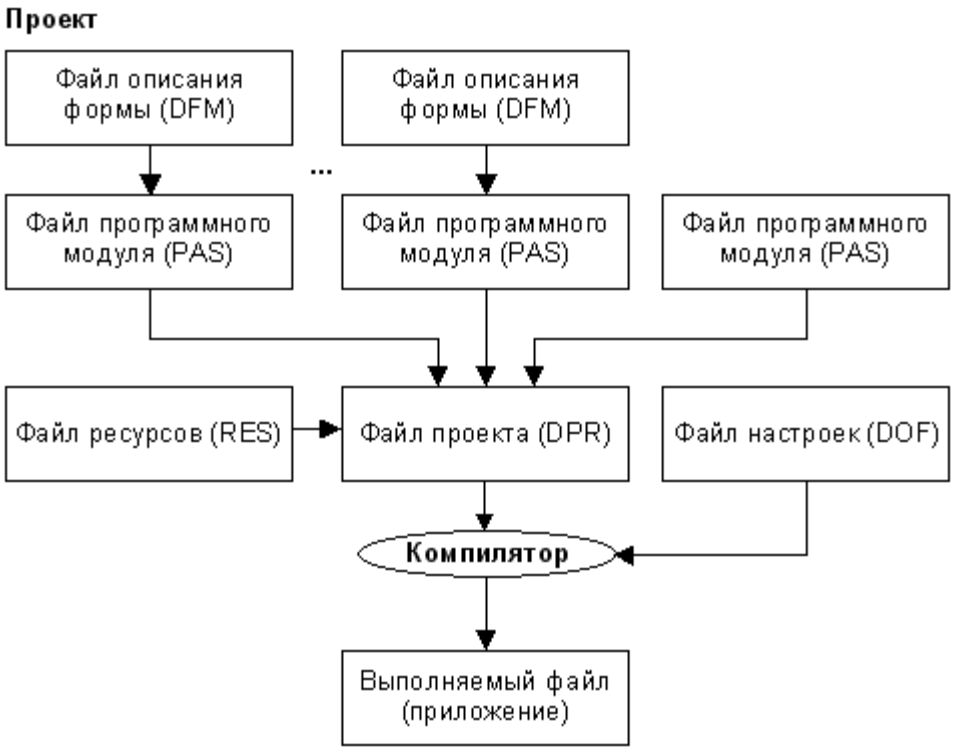


*Рисунок 7.4.Состав проекта в среде Delphi*

Итак, состав проекта понятен. Нужно теперь выяснить, как им управлять — создавать и сохранять проект, добавлять и удалять модули, устанавливать параметры компиляции, собирать и запускать приложение. Этим сейчас и займемся.

## 7.2. Управление проектом

### 7.2.1. Создание, сохранение и открытие проекта

При запуске среды Delphi автоматически создается новый проект. Это сделано для вашего удобства. Если вам потребуется создать новый проект, не перегружая среду Delphi, просто выполните команду меню **File | New | Application**. В результате старый проект будет закрыт, а вместо него создан новый. В новый проект среда Delphi всегда помещает чистую форму.

В процессе разработки приложения вы добавляете на форму компоненты, пишите обработчики событий, добавляете в проект дочерние формы, в общем, проектируете приложение. Когда что-то уже сделано, имеет смысл сохранить проект. Для этого выполните команду главного меню **File | Save All**. Среда запросит сначала имя для программного модуля с формой, а затем имя для проекта (кстати, вы уже сохраняли файл в первой главе). Если файл с введенным именем уже есть на диске, среда Delphi сообщит вам об этом и запросит подтверждение на перезапись существующего файла или запись под другим именем.

Если вдруг потребуется заменить имя проекта другим именем, воспользуйтесь командой меню **File | Save Project As...** . Если же нужно заменить имя модуля, воспользуйтесь командой **File | Save As...** . Операции эти элементарны и не требуют дальнейших пояснений.

Для открытия в среде Delphi ранее сохраненного на диске проекта достаточно выполнить команду главного меню **File | Open...** . На экране появится окно диалога (рисунок 7.5), где вы должны указать или выбрать из представленного списка каталог и имя загружаемого проекта.

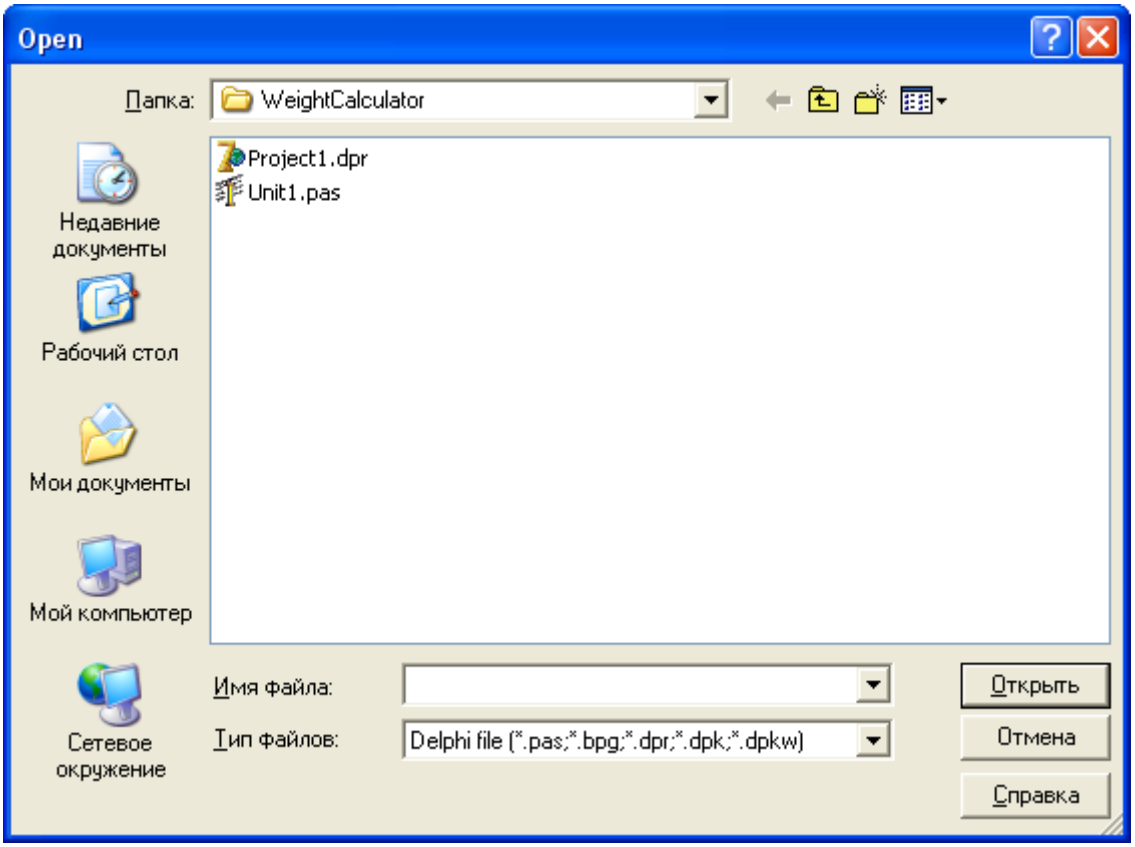


*Рисунок 7.5. Окно выбора проекта*

С открытым проектом можно продолжить работу: исправить, компилировать, выполнить, и не забыть сохранить.

## 7.2.2. Окно управления проектом

При создании более или менее сложного приложения программист всегда должен знать, на какой стадии разработки он находится, иметь представление о составе проекта, уметь быстро активизировать нужный файл, добавить в проект какой-либо новый файл или удалить ненужный, установить параметры компиляции, и т.д. Для этого в среде Delphi имеется *окно управления проектом* — окно **Project Manager**. Фактически это визуальный инструмент для редактирования главного файла проекта. Окно управления проектом вызывается из главного меню командой **View | Project Manager**. После выбора этой команды на экране появится окно, в котором проект представлен в виде дерева (рисунок 7.6).

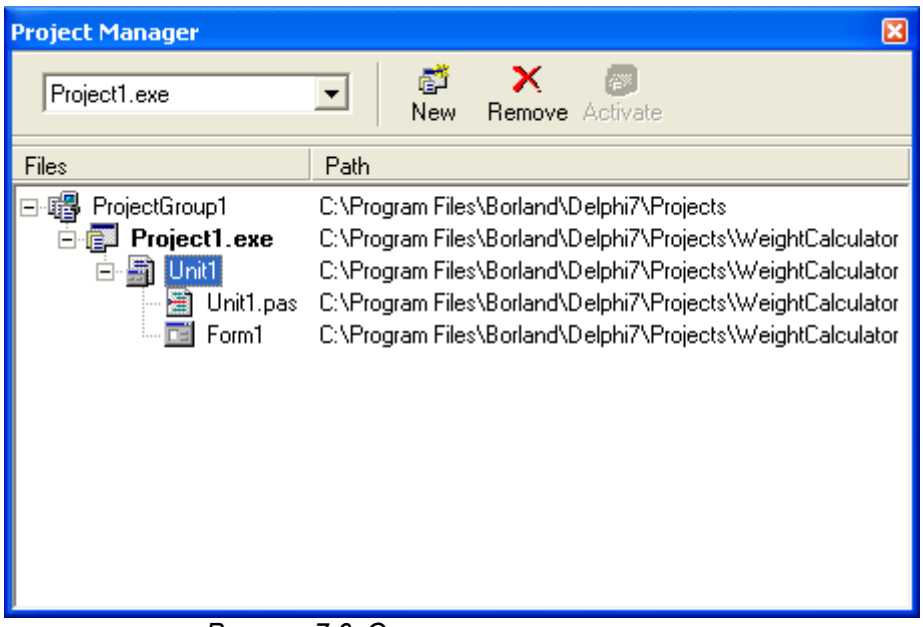


*Рисунок 7.6. Окно управления проектом*

Выделенный жирным шрифтом элемент Project1 — это наш проект. Его имя совпадает с именем выполняемого файла, который получается в результате компиляции и сборки всех модулей проекта. Список модулей отображается в виде подчиненных элементов. Если элемент является формой, то он в свою очередь сам состоит из двух подчиненных элементов: программного модуля формы (PAS-файл) и описания формы (DFM-файл).

Управление проектом выполняется с помощью контекстного меню, которое вызывается щелчком правой кнопки мыши по элементу Project1 (рисунок 7.7).

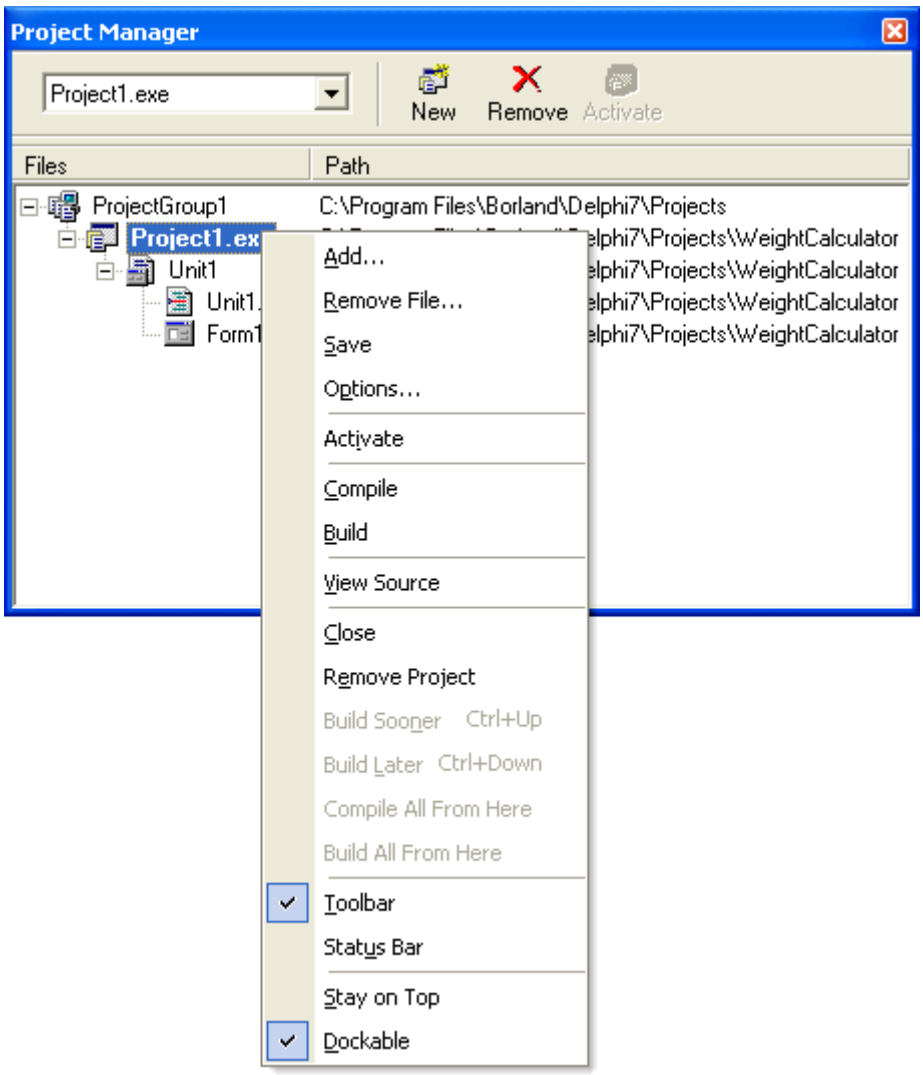


*Рисунок 7.7. Контекстное меню проекта*

Назначение команд контекстного меню кратко описано в следующей таблице:

| Команда | Описание |
|---|---|
| Add… | Добавляет существующий файл (модуль) в проект. |
| Remove File… | Удаляет файл (модуль) из проекта. |
| Save | Сохраняет проект на диск. |
| Options… | Вызывает диалоговое окно настройки проекта (**Project Options**). |
| Activate | Делает проект активным (при работе с группой проектов, см. параграф 7.2.3). |
| Close | Закрывает проект. |
| Remove Project | Удаляет проект из группы (см. параграф 7.2.3). |
| Build Sooner | Перемещает проект вверх по списку, определяющему очередность сборки проектов. Используется при работе с группой проектов (см. параграф 7.2.3). |

| | |
|---|---|
| Build Later | Перемещает проект вниз по списку, определяющему очередность сборки проектов. Используется при работе с группой проектов (см. параграф 7.2.3). |
| Compile All From Here | Компилирует измененные проекты по порядку, начиная с выделенного проекта. Используется при работе с группой проектов (см. параграф 7.2.3). |
| Build All From Here | Компилирует все проекты по порядку, начиная с выделенного проекта. Используется при работе с группой проектов (см. параграф 7.2.3). |

Управление отдельным модулем выполняется с помощью контекстного меню, которое вызывается щелчком правой кнопки мыши по соответствующему элементу, например Unit1 (рисунок 7.8).

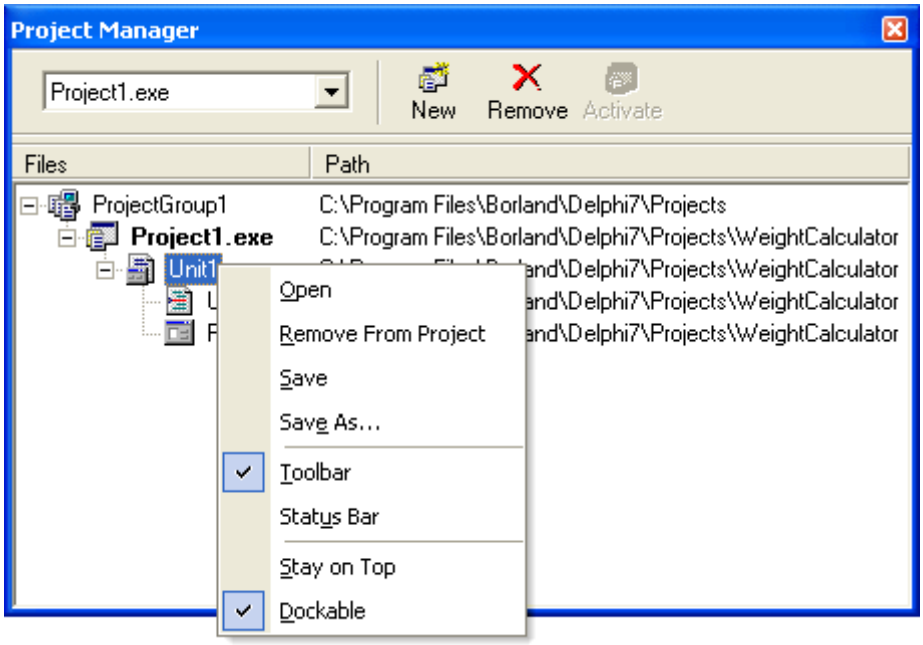


*Рисунок 7.8. Контекстное меню модуля в окне управления проектом*

Назначение основных команд контекстного меню кратко описано в следующей таблице:

| Команда | Описание |
|---|---|
| Open | Открывает модуль. Если модуль содержит форму, то на экране появляется ее графическое представление. Иначе, на экране появляется редактор кода с исходным текстом программного модуля. |
| Remove From Project | Удаляет модуль из проекта. |
| Save | Сохраняет модуль на диск. |
| Save As… | Сохраняет модуль с новым именем. |

Теперь вы всегда сможете узнать, из каких файлов состоит тот или иной проект, а управление им не составит для вас никакого труда.

## 7.2.3. Группы проектов

На практике несколько проектов могут быть логически связаны между собой, например проект динамически подключаемой библиотеки связан с проектом приложения, в котором используется эта библиотека. Среда Delphi позволяет объединить такие проекты в группу. Именно для этого в окне управления проектом имеется корневой элемент ProjectGroup1, подчиненными элементами которого и являются логически связанные проекты. Порядок элементов определяет очередность сборки проектов. Изменить порядок можно с помощью команд **Build Sooner** и **Build Later**, которые находятся в контекстном меню, вызываемом щелчком правой кнопки мыши по соответствующему проекту (рисунок 7.7).

На данный момент в группе существует только один проект — Project1. Для добавления других проектов в группу воспользуйтесь контекстным меню, которое вызывается щелчком правой кнопки мыши по элементу ProjectGroup1 (рисунок 7.9).

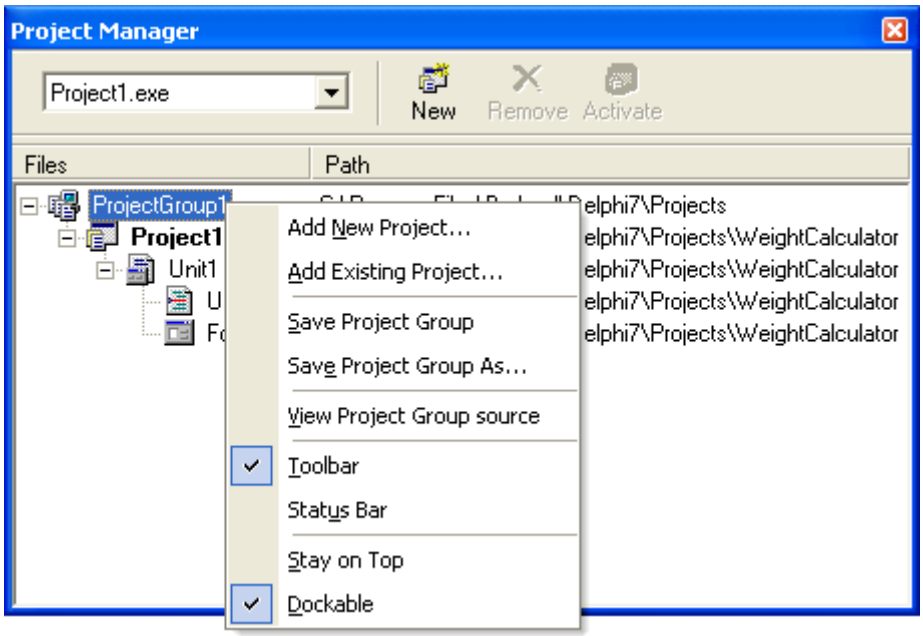


*Рисунок 7.9. Контекстное меню группы проектов*

Назначение команд контекстного меню кратко описано в следующей таблице:

| Команда | Описание |
|---|---|
| Add New Project… | Создает новый проект и добавляет его в группу. |
| Add Existing Project… | Добавляет существующий проект в группу. |
| Save Project Group | Сохраняет файл, описывающий группу проектов. |
| Save Project Group As… | Сохраняет описание группы проектов в файле с другим именем. |
| View Project Group source | Показывает текстовый файл, описывающий группу проектов. |

Когда в группу объединены несколько проектов, среда Delphi создает специальный текстовый файл с описанием этой группы. Файл имеет расширение BPG (от англ. Borland Project Group), а его имя запрашивается у пользователя. Для групп, состоящих из одного единственного проекта BPG-файл не создается.

## 7.2.4. Настройка параметров проекта

Проект имеет много различных параметров, с помощью которых вы управляете процессом компиляции и сборки приложения. Установить параметры проекта можно в окне **Project Options**. Для этого выберите в главном меню команду **Project | Options...** или в окне управления проектом вызовите контекстное меню элемента Project1 и выберите команду **Options...**. На экране появится диалоговое окно; вам останется лишь установить в нем нужные значения параметров.

Диалоговое окно параметров проекта состоит из нескольких вкладок. Параметров очень много, поэтому мы рассмотрим только те, которые используются наиболее часто.

На вкладке **Forms** (рисунок 7.10) можно задать главную форму приложения (**Main form**) и в списке **Auto-create forms** указать формы, которые будут создаваться одновременно с главной формой.

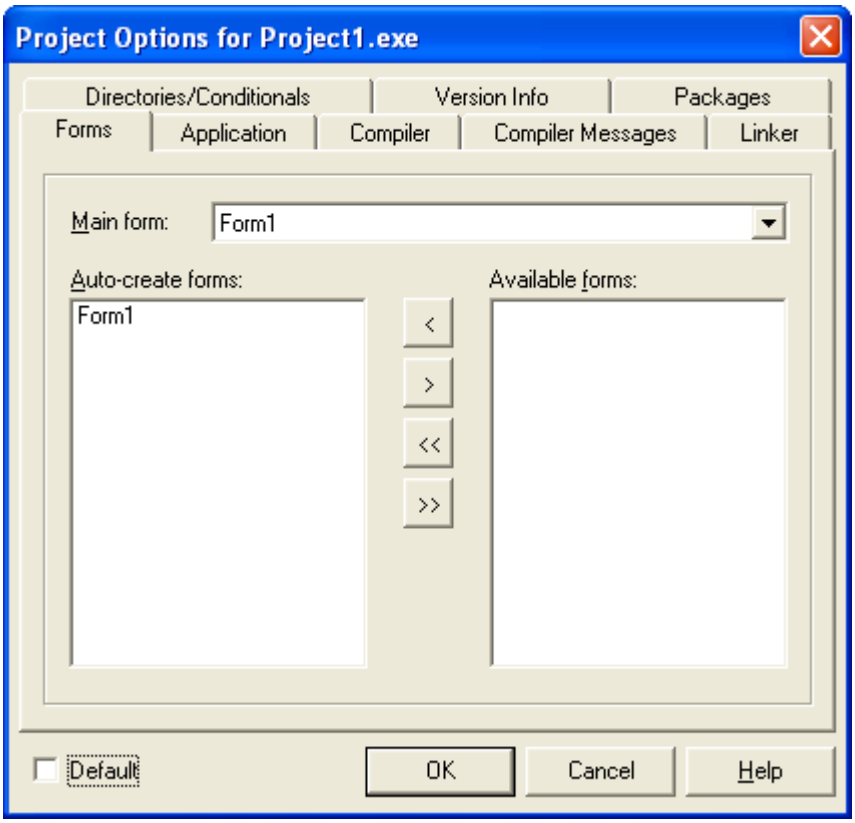


*Рисунок 7.10. Окно параметров проекта. Вкладка Forms*

На вкладке **Application** (рисунок 7.11) можно задать название (**Title**) вашей программы. В среде Delphi дополнительно можно задать файл справки (**Help file**) и значок (**Icon**).

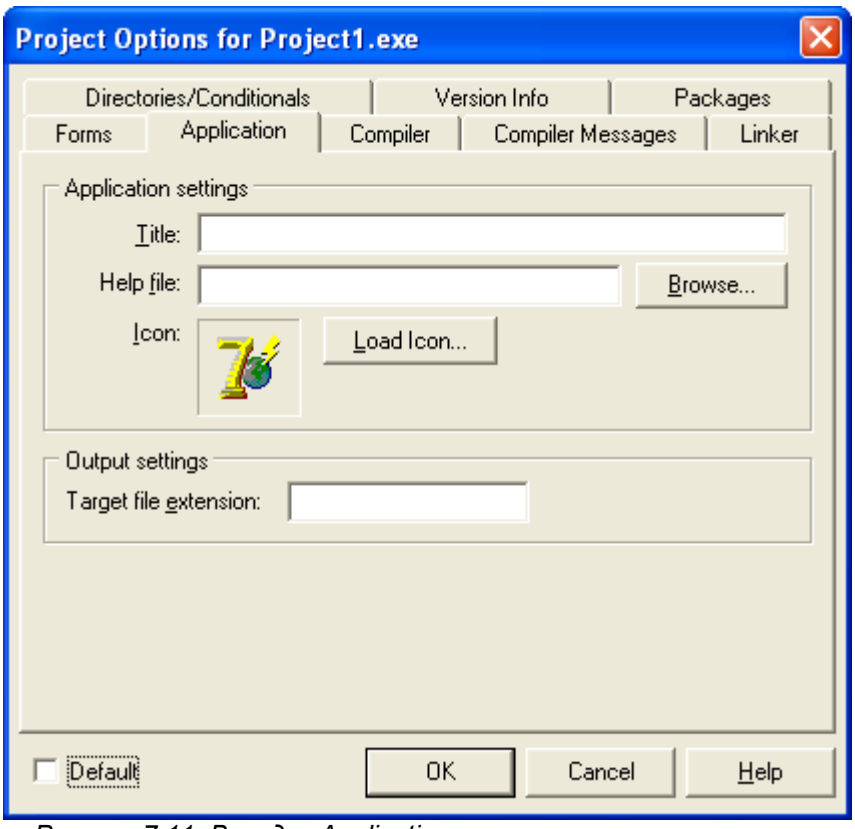


*Рисунок 7.11. Вкладка Application в окне параметров проекта*

На вкладке **Compiler** (рисунок 7.12) настраиваются параметры компилятора. Наиболее интересными из них являются переключатели **Optimization** (включает оптимизацию генерируемого кода) и **Use Debug DCUs** (позволяет отлаживать исходный код системных библиотек). Оба этих переключателя полезны при отладке программы: первый следует выключить, а второй — включить.

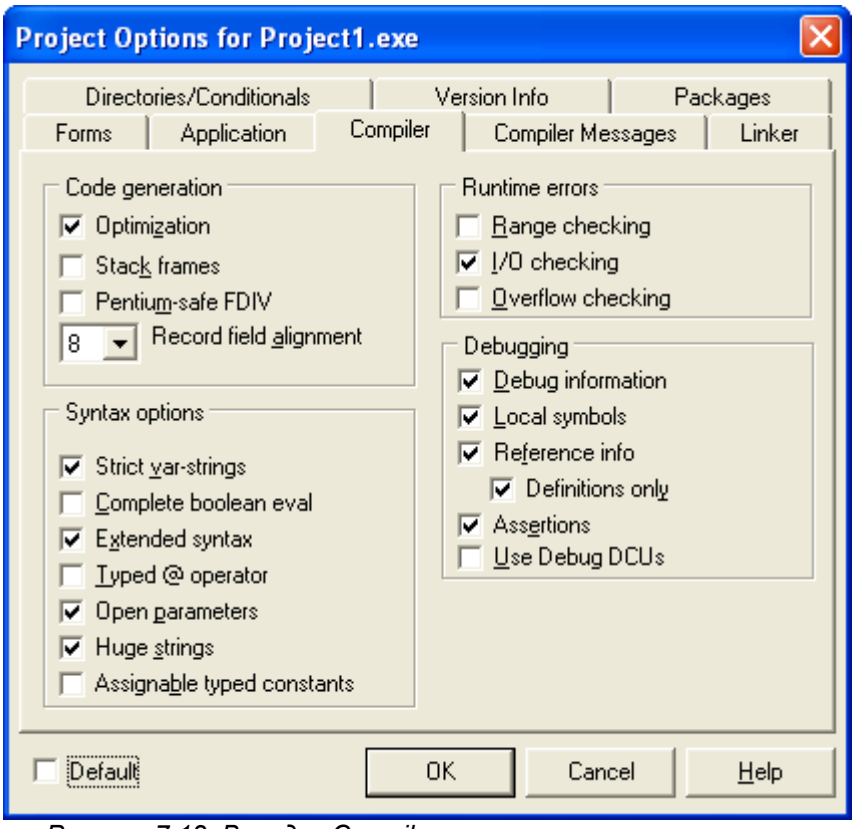


*Рисунок 7.12. Вкладка Compiler в окне параметров проекта*

На вкладке **Compiler Messages** (рисунок 7.13) настраивается чувствительность компилятора к подозрительному коду. Включив переключатели **Show hints** и **Show warnings**, вы будете получать от компилятора весьма полезные подсказки (hints) и предупреждения (warnings), а не только сообщения об ошибках.

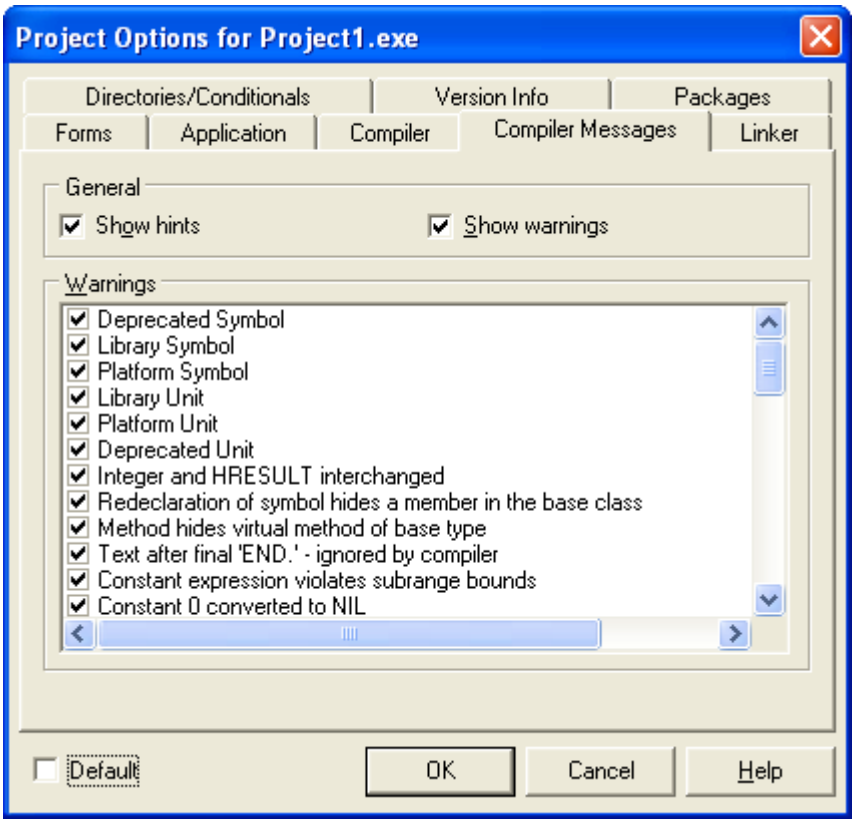


*Рисунок 7.13. Вкладка Compiler Messages в окне параметров проекта*

На вкладке **Linker** (рисунок 7.14) настраиваются параметры сборки проекта. Обладателям среды Delphi следует обратить внимание на группу **Memory sizes**, особенно на два параметра: **Min stack size** и **Max stack size**. Они задают соответственно минимальный и максимальный размеры стека прикладной программы. Вам может потребоваться увеличить значения этих параметров при написании приложения, активно использующего рекурсивные подпрограммы.

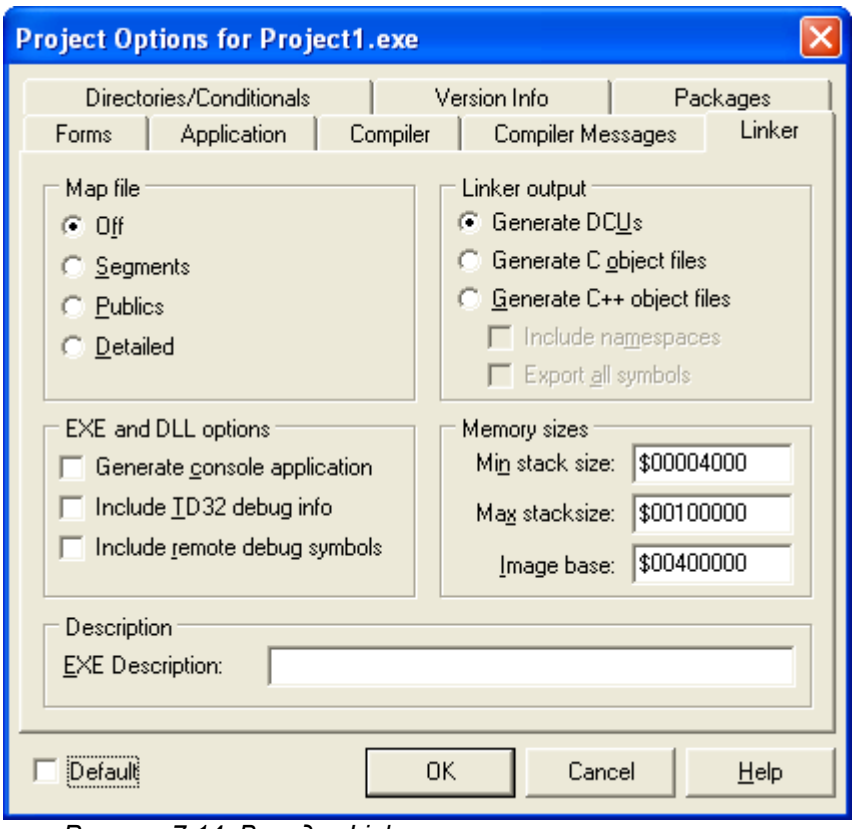


*Рисунок 7.14. Вкладка Linker в окне параметров проекта*

На вкладке **Directories/Conditionals** (рисунок 7.15) можно задать каталоги для различных файлов. Наиболее важные из них: **Output directory** — каталог, в который помещается выполняемый файл; **Unit output directory** — каталог, в который помещаются промежуточные объектные модули (DCU-файлы); **Search path** — список каталогов, в которых осуществляется поиск программных модулей.

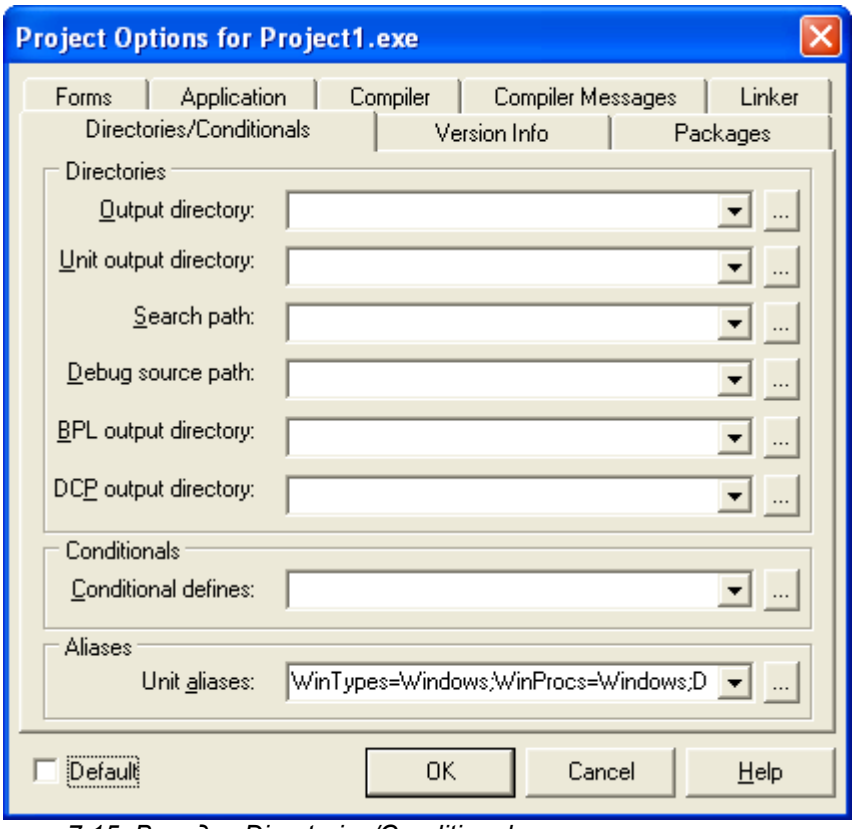


*Рисунок 7.15. Вкладка Directories/Conditionals в окне параметров проекта*

На вкладке **Version Info** (рисунок 7.16) выводится информация о версии приложения. Для того чтобы эта информация помещалась в выполняемый файл, нужно включить переключатель **Include version information in project**. Номер версии задается в виде четырех чисел: **Major version** — старший номер версии программы (его обычно увеличивают при внесении в программу концептуально новых возможностей); **Minor version** — младший номер версии программы (его обычно увеличивают при незначительном расширении функциональных возможностей программы); **Release** — номер выпуска программы, которая отлажна и пригодна к использованию заказчиком; **Build** — порядковый номер сборки проекта (он автоматически увеличивается на единицу при компиляции проекта, если включен переключатель **Auto-increment build number**). Все эти параметры несут лишь информативный характер и не влияют на работу самой программы. Однако, информация о версии может использоваться специальными программами установки пользовательских программ для контроля за тем, чтобы более новые версии библиотек не заменялись более старыми.

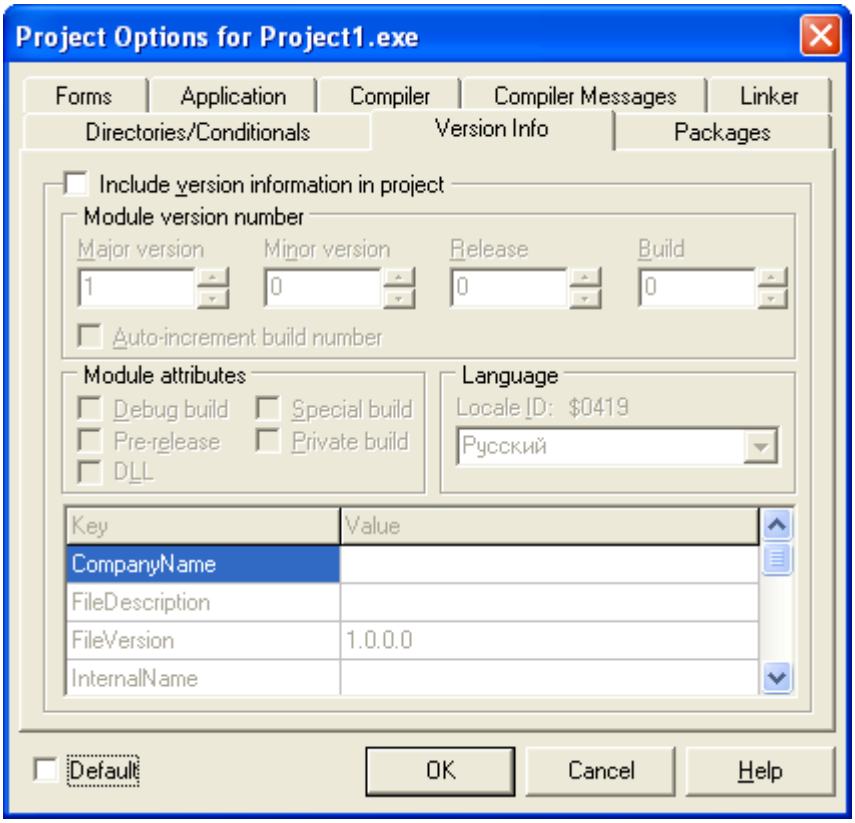


*Рисунок 7.16. Вкладка Version Info в окне параметров проекта*

На вкладке **Packages** (рисунок 7.17) вы можете управлять списком пакетов, используемых в вашем проекте. *Пакеты* — это внешние библиотеки компонентов, они рассмотрены в главе 13. Обратите внимание на переключатель **Build with runtime packages**, который позволяет существенно уменьшить размер выполняемого файла за счет использования внешних библиотек компонентов вместо встраивания их кода непосредственно в выполняемый файл. Этот режим выгоден при создании нескольких программ, построенных на базе большого количества общих компонентов.

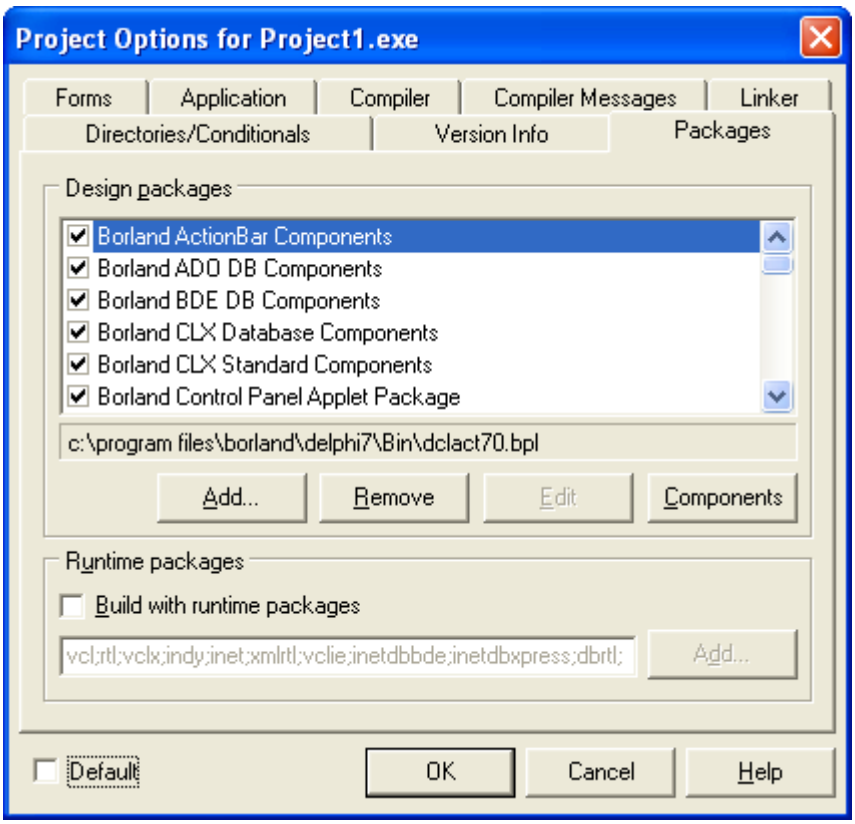


*Рисунок 7.17. Вкладка Packages в окне параметров проекта*

Когда все параметры проекта установлены, можно приступать к его компиляции.

## 7.2.5. Компиляция и сборка проекта

Компиляция и сборка проекта могут выполняться на любой стадии разработки проекта. Под *компиляцией* понимается получение объектных модулей (DCU-файлов) из исходных текстов программных модулей (PAS-файлов). Под *сборкой* понимается получение выполняемого файла из объектных модулей. В среде Delphi компиляция и сборка проекта совмещены.

Для выполнения компиляции достаточно выполнить команду меню **Project | Compile <Имя проекта>** или нажать комбинацию клавиш Ctrl+F9. При этом компилируются все исходные модули, содержимое которых изменялось после последней компиляции. В результате для каждого программного модуля создается файл с расширением DCU (сокр. от Delphi Compiled Unit). Затем среда Delphi компилирует главный файл проекта и собирает (иногда говорят компонует) из DCU-модулей выполняемый файл, имя которого совпадает с именем проекта. К сведению профессионалов заметим, что смышленый компилятор среды Delphi выбрасывает из выполняемого файла весь неиспользуемый программный код, поэтому не стоит волноваться по поводу лишних объектов и подпрограмм, которые могут присутствовать в подключенных модулях.

Существует особый вид компиляции и сборки — полная принудительная компиляция всех программных модулей проекта, для которых доступны исходные тексты, с последующей сборкой выполняемого файла. При этом не важно, вносились в них изменения после предыдущей компиляции или нет. Полная компиляция проекта выполняется с помощью команды главного меню **Project | Build <Имя проекта>**. В результате тоже создается выполняемый файл, но на это тратиться немного больше времени.

## 7.2.6. Запуск готового приложения

Когда после многочисленных компиляций вы исправите все ошибки и получите-таки выполняемый файл, можно будет посмотреть на результат вашего самоотверженного труда. Для этого надо выполнить созданное приложение с помощью команды меню **Run | Run** или клавиши F9. Перед выполнением будет автоматически повторен процесс компиляции (если в проект

вносились изменения) и после его успешного завершения приложение запустится на выполнение. В результате вы увидите на экране его главную форму.

Вот собственно и все, что мы хотели поведать вам о проекте. В целом вы представляете, что такое проект, и знаете, как им управлять. Пора заняться составными частями проекта и одновременно основными элементами любого приложения в среде Delphi — формами.

# 7.3. Форма

## 7.3.1. Понятие формы

Из первой главы вы уже имеете общее представление о форме, теперь настало время изучить ее более пристально. Фактически форма — это главный компонент приложения, который, как и менее значительные компоненты, имеет свойства. Важнейшие свойства формы: заголовок, высота, ширина, местоположение, цвет фона и др. При создании новой формы среда Delphi сама задает начальные значения свойствам формы, но вы можете изменить их так, как считаете нужным. Это можно сделать во время проектирования формы (в окне свойств) или во время выполнения приложения (с помощью операторов языка Delphi).

Форма имеет очень много свойств, и поначалу в них легко запутаться. Практика показывает, что путаница возникает из-за алфавитного порядка свойств в окне **Object Inspector**: близкие по смыслу свойства оказываются разбросанными по ячейкам списка. Чтобы у вас сложилось представление о возможностях формы, рассмотрим основные свойства формы в порядке их важности. Для этого нам понадобится новое приложение.

Выберите в меню команду **File | New | Application**. Среда Delphi автоматически создаст в новом проекте чистую форму и поместит ее исходный текст в редактор кода (рисунок 7.18).

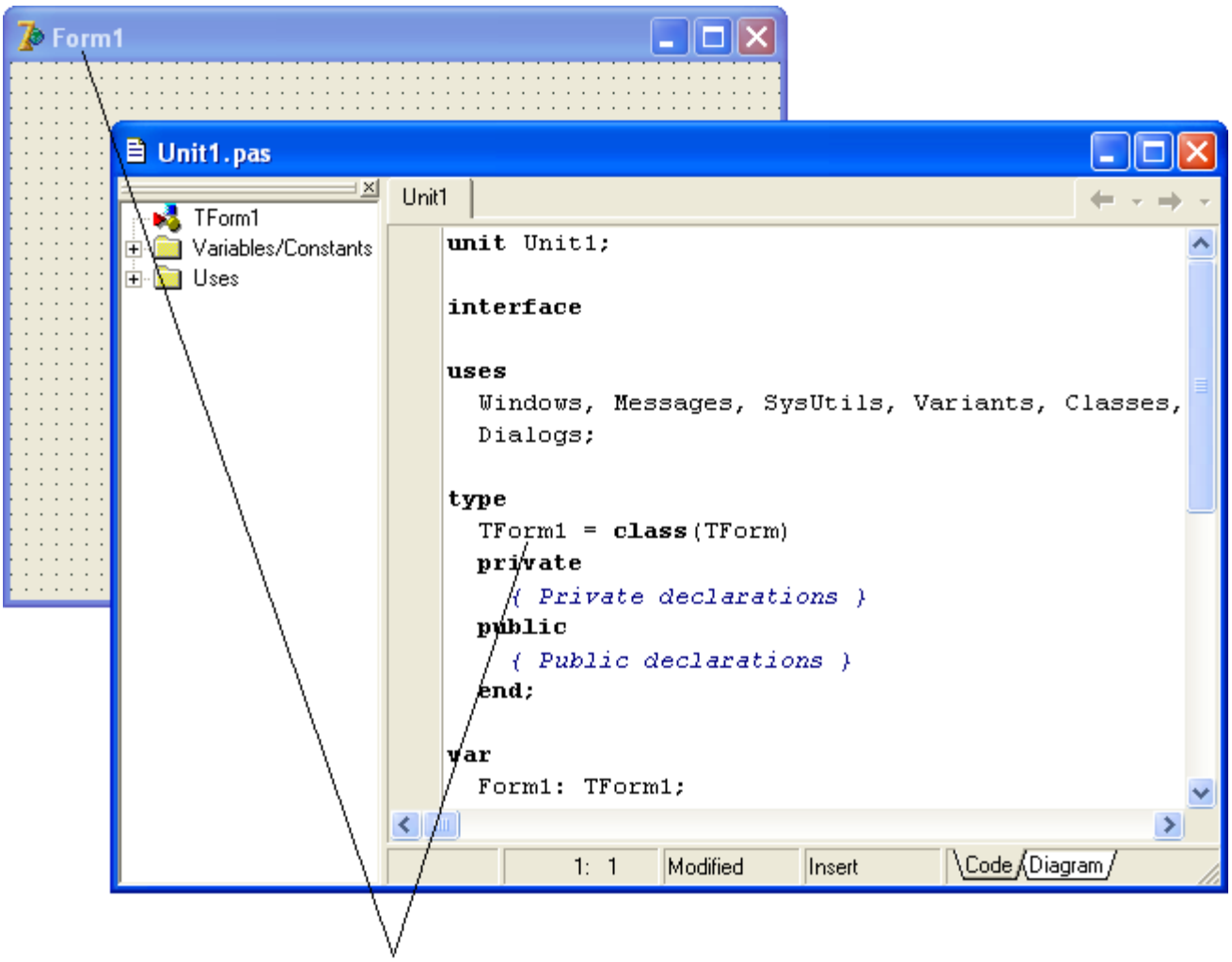


*Рисунок 7.18. Форма на экране и ее описание в редакторе кода*

Сразу сохраните проект и его форму, чтобы дать им осмысленные имена. Выберите в меню команду **File | Save All** и дайте модулю имя **Main.pas**, а проекту — имя **FormTest.dpr**. Полигон для изучения формы подготовлен, можно заняться ее свойствами.

## 7.3.2. Имя и заголовок формы

Главное свойство, с которого вы начинаете настройку формы, – это свойство **Name** — имя. Оно содержит идентификатор, используемый для обращения к форме из программы (рисунок 7.19). Первой же форме нового проекта автоматически назначается имя **Form1**. Мы советуем всегда его изменять, чтобы имя формы отражало ее роль в приложении. Например, главную форму приложения можно назвать **MainForm** (если ничего лучше в голову не приходит).

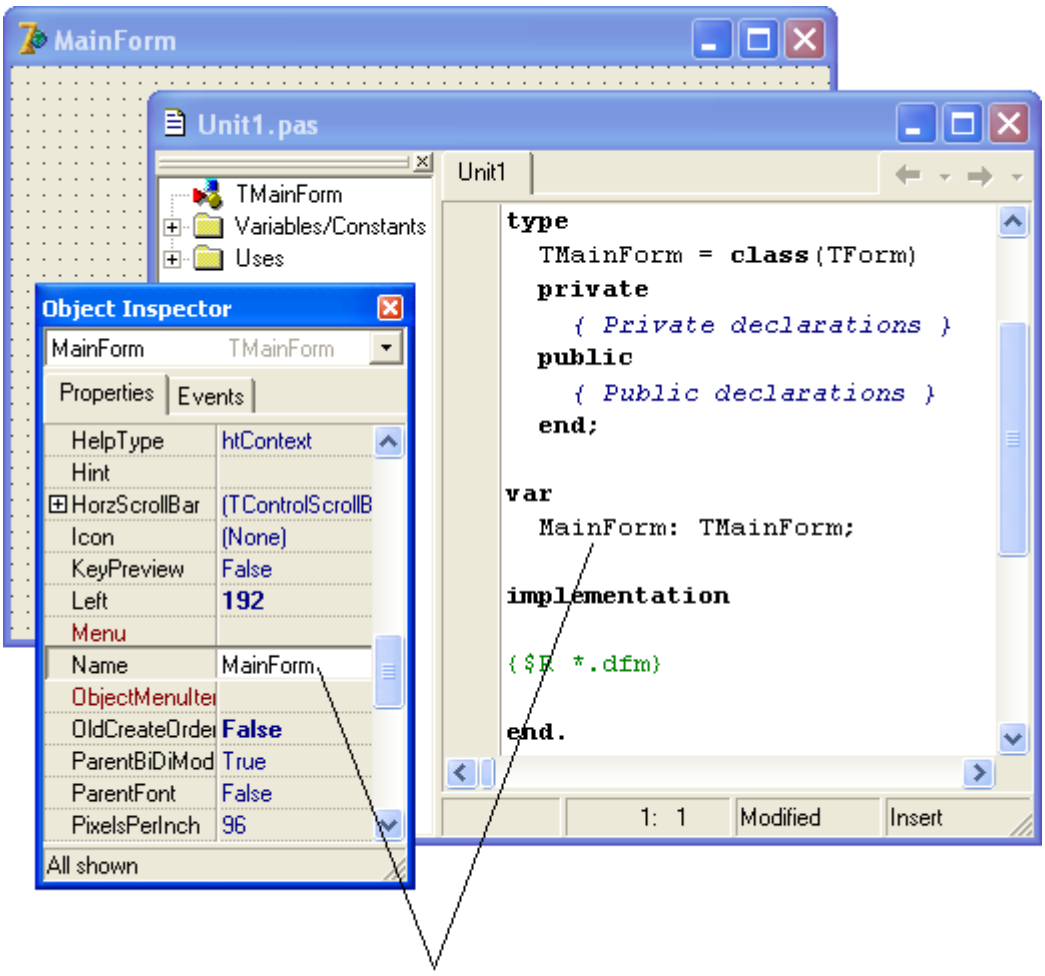


*Рисунок 7.19. Программный идентификатор формы*

На будущее заметим, что свойство **Name** есть в любом компоненте, и оно редактируется в окне свойств.

Каждая форма приложения должна иметь понятный заголовок, говорящий пользователю о ее назначении. Заголовок задается в свойстве **Caption**. Наша форма — учебная, поэтому мы дадим ей заголовок **Main**, говорящий о том, что это просто главная форма (рисунок 7.20).

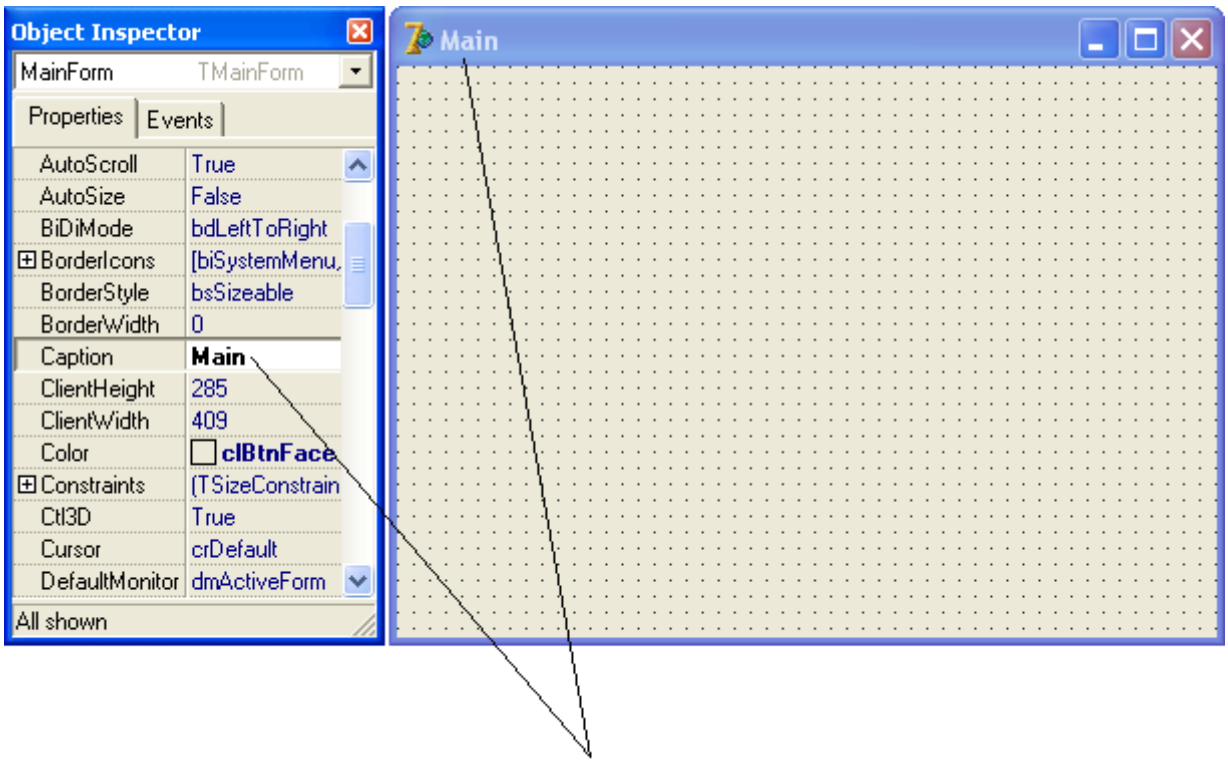


*Рисунок 7.20. Заголовок формы*

**ПРИМЕЧАНИЕ**

Внимание! Если вам вдруг взбрела в голову идея изменить шрифт, цвет или высоту заголовка, то не ищите для этого подходящих свойств. Все это — параметры графической оболочки операционной системы, и задаются они в настройках этой оболочки.

## 7.3.3. Стиль формы

Настраивая форму, нужно принимать во внимание, какой пользовательский интерфейс будет иметь ваше приложение: многодокументный интерфейс MDI (от англ. Multiple Document Interface) или обычный одно-документный интерфейс SDI (от англ. Single Document Interface). За это отвечает свойство формы **FormStyle**, которое может принимать следующие значения:

- fsMDIChild – дочернее окно MDI-приложения;
- fsMDIForm – главное окно MDI-приложения;
- fsNormal – обычное окно (значение по умолчанию);
- fsStayOnTop – окно, всегда расположенное поверх других окон на экране.

Многие приложения, с которыми вы работаете, имеют пользовательский интерфейс MDI. Они состоят из основного окна, которое включает одно или несколько внутренних окон. Внутренние окна ограничены областью основного окна и не могут выходить за его границы. Для главной формы, соответствующей основному окну MDI-приложения, значение свойства **FormStyle** должно быть равно fsMDIForm. Для всех второстепенных форм, соответствующих внутренним окнам, значение свойства **FormStyle** равно fsMDIChild. Для окон диалога, выполняющихся в монопольном режиме, свойство **FormStyle** равно значению fsNormal, что дает возможность выносить их за пределы основной формы.

Если программа имеет пользовательский интерфейс SDI, то каждая форма существует в виде отдельного независимого окна. Одно из окон является главным, однако оно не содержит другие окна. В SDI-приложении значение свойства **FormStyle** равно fsNormal и для главной формы, и для второстепенных форм. В некоторых случаях допускается установка значения fsStayOnTop для того, чтобы форма всегда отображалось поверх других форм.

Очевидно, что наш простой вычислитель идеального веса является SDI-приложением и поэтому свойство **FormStyle** имеет значение по умолчанию — fsNormal.

### 7.3.4. Размеры и местоположение формы на экране

Теперь определимся с размерами формы и ее местоположением на экране. Установить размеры и положение формы проще всего во время проектирования с помощью мыши. Другой способ — обратиться к окну свойств и задать размеры формы с помощью свойств **Width** и **Height**, а местоположение — с помощью свойств **Left** и **Top** (значения задаются в пикселах). Смысл свойств поясняет рисунок 7.21.

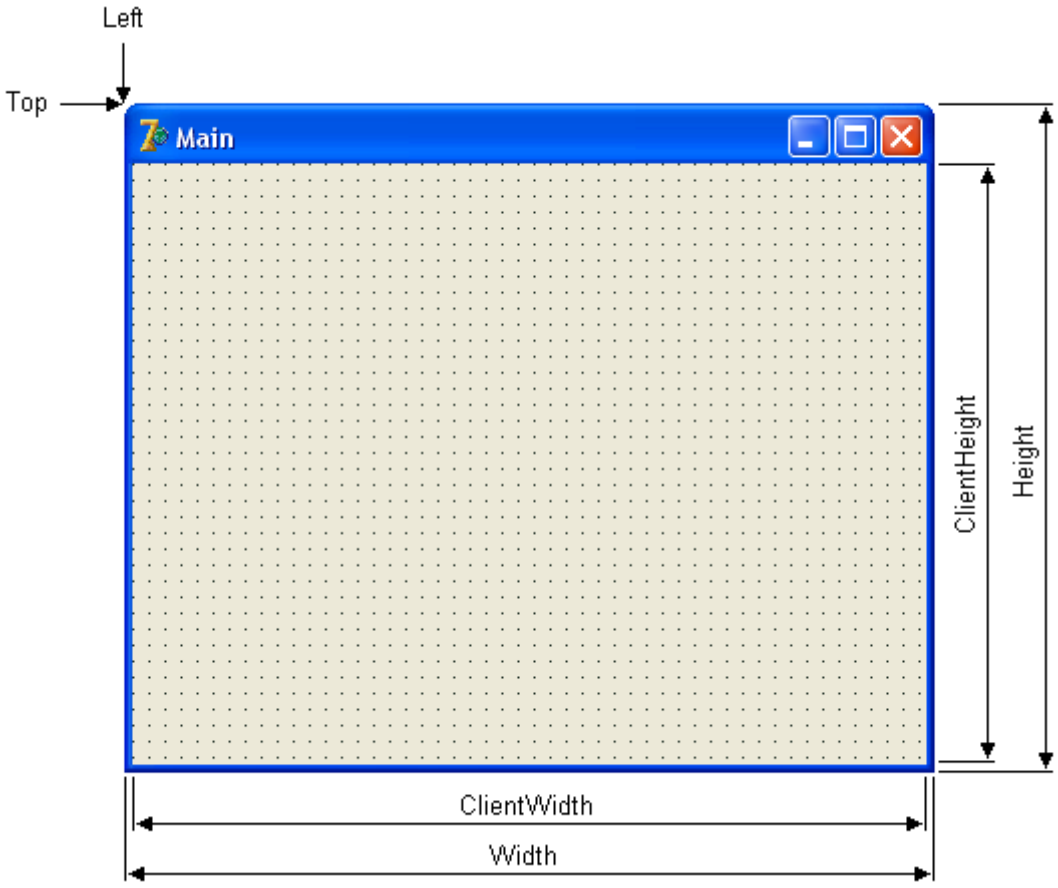


*Рисунок 7.21. Размеры и местоположение формы на экране*

Кроме того, с помощью свойства **Position** можно организовать автоматическое размещение формы на экране, выбрав одно из следующих возможных значений:

- poDefault – размеры и положение формы подбираются автоматически исходя из размеров экрана.
- poDefaultPosOnly – положение формы подбирается автоматически, а ширина и высота определяются значениями свойств **Width** и **Height** соответственно.
- poDefaultSizeOnly – размеры формы устанавливаются автоматически, а местоположение определяется значениями свойств **Left** и **Top**.
- poDesigned – размеры и положение формы определяются значениями свойств **Left**, **Top**, **Width**, **Height**.
- poDesktopCenter – форма размещается в центре рабочего стола (т.е. экрана, из которого исключена панель задач). Размеры формы определяются значениями свойств **Width** и **Height**.
- poMainFormCenter – форма центрируется относительно главной формы. Размеры формы определяются значениями свойств **Width** и **Height**.
- poOwnerFormCenter – форма центрируется относительно формы-владельца. Размеры формы определяются значениями свойств **Width** и **Height**.
- poScreenCenter – форма размещается в центре экрана. Размеры формы определяются значениями свойств **Width** и **Height**.

Иногда размеры формы рассчитываются исходя из размеров ее внутренней *рабочей области* (client area), на которой размещаются компоненты. Как известно, в рабочую область не входят рамка и заголовок. Размеры рабочей области хранятся в свойствах **ClientWidth** и **ClientHeight**. При их установке значения свойств **Width** и **Height** автоматически пересчитываются (и наоборот).

Бывает, что при выборе размеров формы учитываются размеры экрана. Поскольку современные видео-адаптеры поддерживают множество режимов с различными разрешениями, встает вопрос: как обеспечить одинаковую пропорцию между формой и экраном независимо от разрешающей способности дисплея. На этот случай в форме предусмотрено свойство **Scaled**. Если оно установлено в значение True, то форма будет автоматически масштабироваться в зависимости от разрешающей способности дисплея.

При перемещении по экрану, форма может слегка прилипать к краям экрана, если края формы находятся в непосредственной близости от них. Это происходит в том случае, если свойство **ScreenSnap** содержит значение True. Расстояние формы до краев экрана, при котором форма прилипает, задается в свойстве **SnapBuffer** и измеряется в пикселях.

Работая с приложением, пользователь может свернуть форму или развернуть ее на всю рабочую область экрана с помощью соответствующих кнопок рамки. Состояние формы (свернута или развернута) определяется свойством **WindowState**, которое принимает следующие значения:

- wsNormal – форма находится в нормальном состоянии (ни свернута, ни развернута на весь экран);
- wsMinimized – форма свернута;
- wsMaximized – форма развернута на весь экран.

Если при проектировании вы измените значение свойства **WindowState** на wsMinimized или wsMaximized, то получите форму, которая при первом появлении будет автоматически либо свернута в панель задач, либо развернута на весь экран.

На компьютере с двумя и более мониторами существует возможность выбрать для формы монитор, на котором она отображается. Для этого следует установить свойство **DefaultMonitor** в одно из следующих значений:

- dmDesktop – форма отображается на текущем мониторе; никаких попыток разместить форму на каком-то конкретном мониторе не делается;
- dmPrimary – форма отображается на первом мониторе в списке **Monitors** объекта **Screen** (см. параграф 7.7.2);
- dmMainForm – форма отображается на том мониторе, на котором находится главная форма;
- dmActiveForm – форма отображается на том мониторе, на котором находится активная в данный момент форма.

Свойство **DefaultMonitor** учитывается лишь в том случае, если в программе существует главная форма.

## 7.3.5. Цвет рабочей области формы

С размерами формы все ясно и теперь желающие могут изменить стандартный цвет ее фона с помощью свойства **Color**. Для этого следует обратиться к окну свойств. Перейдите к свойству **Color**, откройте выпадающий список, и выберите любой цвет из набора базовых цветов. Базовые цвета представлены в списке именованными константами. Вы можете также выбрать цвет из всей цветовой палитры, выполнив двойной щелчок мыши в поле значения свойства. На экране появится стандартное диалоговое окно выбора цвета (рисунок 7.22).

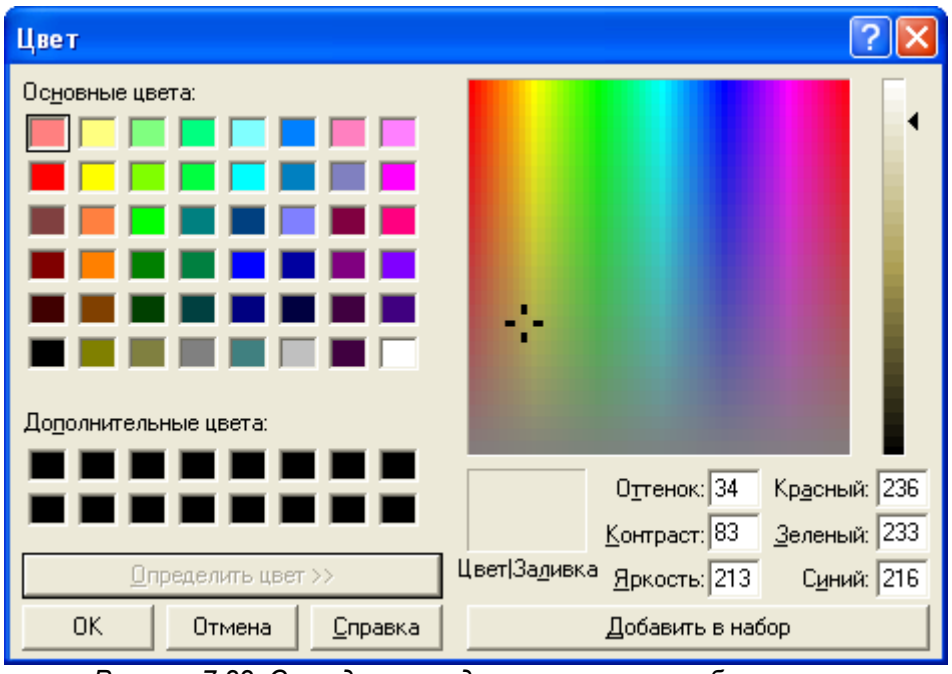


*Рисунок 7.22. Стандартоне диалоговое окно выбора цвета*

Когда вы установите цвет в окне свойств, изменение немедленно отразится на форме. Можно работать с самыми разными цветами, но хорошим тоном считается использовать стандартную цветовую гамму. Поэтому лучшее значение для свойства **Color** — clBtnFace (цвет такой, как у кнопок).

## 7.3.6. Рамка формы

Во внешнем виде формы очень важную роль играет рамка и расположенные на ней кнопки "Свернуть", "Развернуть", "Закрыть" (рисунок 7.23). Стиль рамки задается с помощью свойства **BorderStyle**, которое может принимать следующие значения:

- bsNone — у окна вообще нет ни рамки, ни заголовка;
- bsDialog — неизменяемая в размерах рамка, свойственная диалоговым окнам;
- bsSingle — неизменяемая в размерах рамка для обычного окна;
- bsSizeable — изменяемая в размерах рамка для обычного окна;
- bsToolWindow — аналогично значению bsSingle, но окно имеет слегка уменьшенный заголовок, что свидетельствует о его служебном назначении;
- bsSizeToolWin — аналогично значению bsSizeable, но окно имеет слегка уменьшенный заголовок, что свидетельствует о его служебном назначении.

Обычно свойство **BorderStyle** имеет значение bsSizeable. В этом случае форма имеет стандартную изменяемую в размерах рамку (как при проектировании), заголовок, меню управления, кнопки "Свернуть", "Развернуть", "Закрыть" и, иногда, "Справка". Для указания того, какие именно из этих элементов отображать, используется свойство **BorderIcons**. Список его возможных значений следующий:

- biSystemMenu — рамка формы содержит меню управления, которое вызывается щелчком правой кнопки мыши по заголовку формы;
- biMinimize – рамка формы имеет кнопку "Свернуть";
- biMaximize – рамка формы имеет кнопку "Развернуть";
- biHelp – рамка формы имеет кнопку "Справка". При нажатии кнопки "Справка", курсор мыши превращается в стрелку со знаком вопроса. Выбирая таким курсором нужный элемент формы, пользователь получает по нему справку во всплывающем окне.

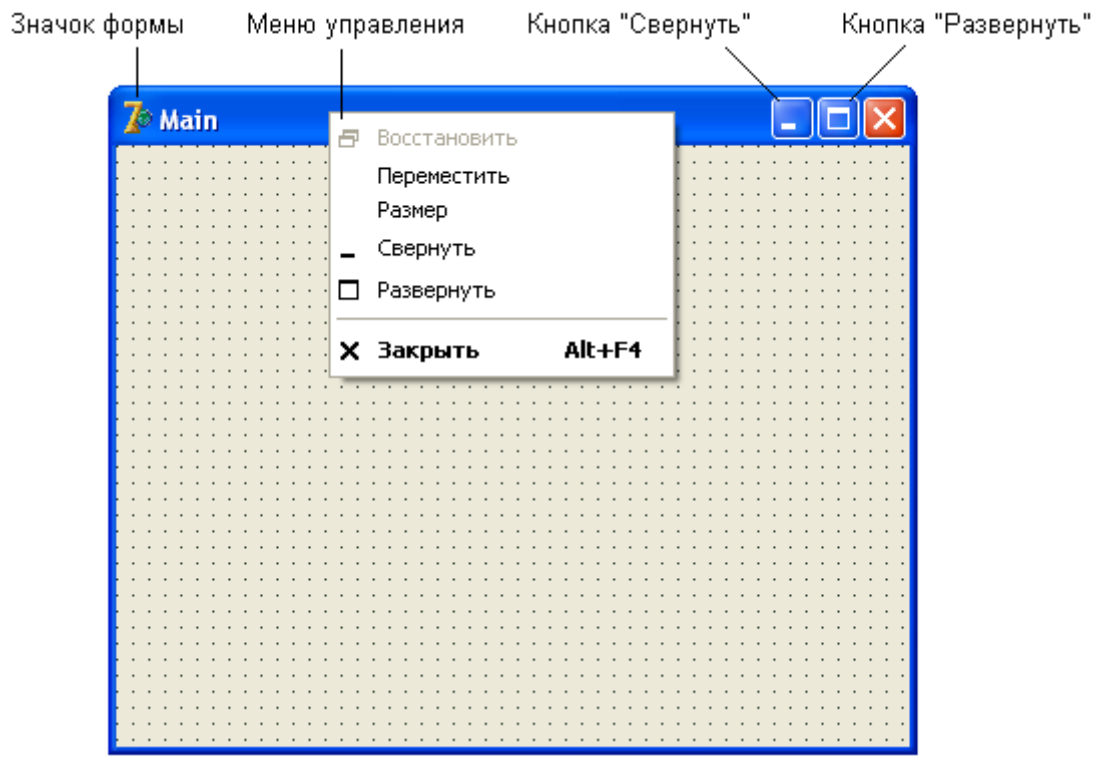


*Рисунок 7.23. Рамка формы и ее контекстное меню*

Команды меню управления не нуждаются в комментариях.

## 7.3.7. Значок формы

Если вы разрабатываете коммерческое приложение, а не тестовый пример, следует позаботиться о том, чтобы форма имела в своем левом верхнем углу выразительный значок. Для разработки значков существует множество средств, на которых мы не будем останавливаться. Когда значок готов и сохранен в файле, его нужно просто установить в качестве значения свойства **Icon** (рисунок 7.24).

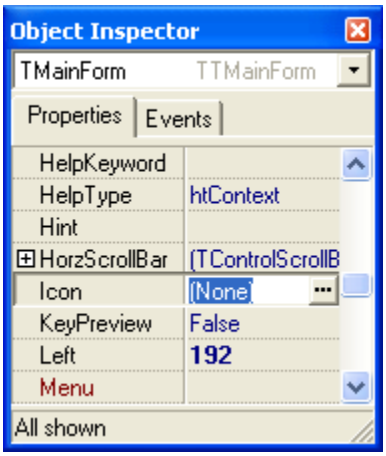


*Рисунок 7.24. Установка значка формы*

Для этого откройте окно **Picture Editor** нажатием кнопки с многоточием (рисунок 7.25). Нажмите кнопку **Load...** и выберите какой-нибудь файл из стандартной коллекции значков (как правило, каталог C:\Program Files\Common Files\Borland Shared\Images). После этого закройте диалоговое окно с помощью кнопки **OK**.

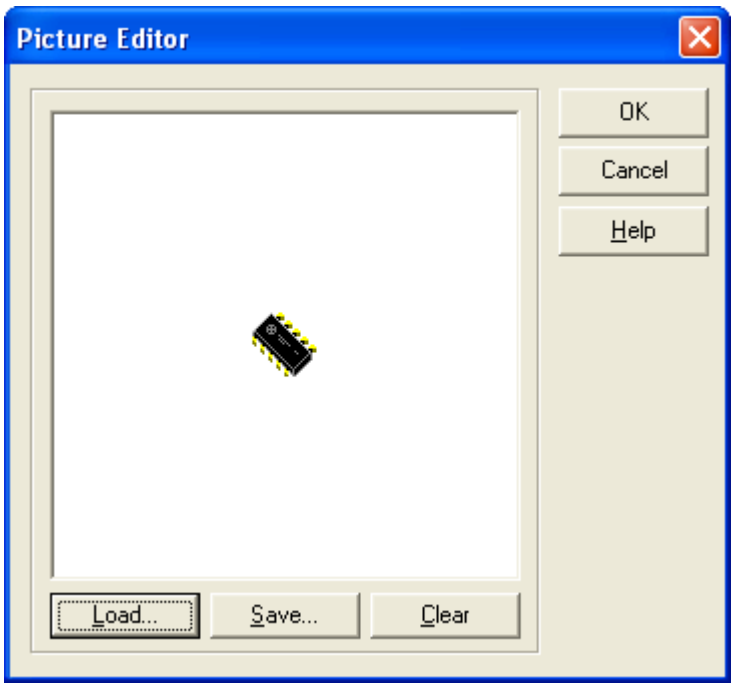


*Рисунок 7.25. Окно выбора значка для формы*

Среда Delphi сразу же подставит выбранный значок в левый верхний угол формы (рисунок 7.26).

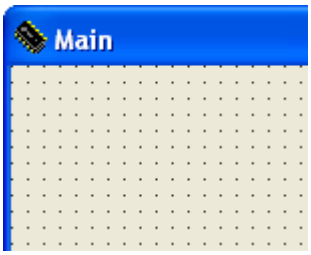


*Рисунок 7.26. Новый значок формы*

### 7.3.8. Невидимая форма

Сценарий решения задачи может потребовать, чтобы в некоторый момент форма стала невидимой, т.е. исчезла с экрана. За "видимость" формы отвечает булевское свойство **Visible**. Установка ему значения False скроет форму, а установка значения True покажет ее.

### 7.3.9. Прозрачная форма

Некоторые части формы можно сделать прозрачными (рисунок 7.27). Причем щелчок мыши по прозрачной области формы будет приводить к активизации окон (форм), находящихся за формой. Это достигается установкой свойства **TransparentColor** в значение True и выбором цвета прозрачности в свойстве **TransparentColorValue**. Все пиксели формы с цветом **TransparentColorValue** будут прозрачными.

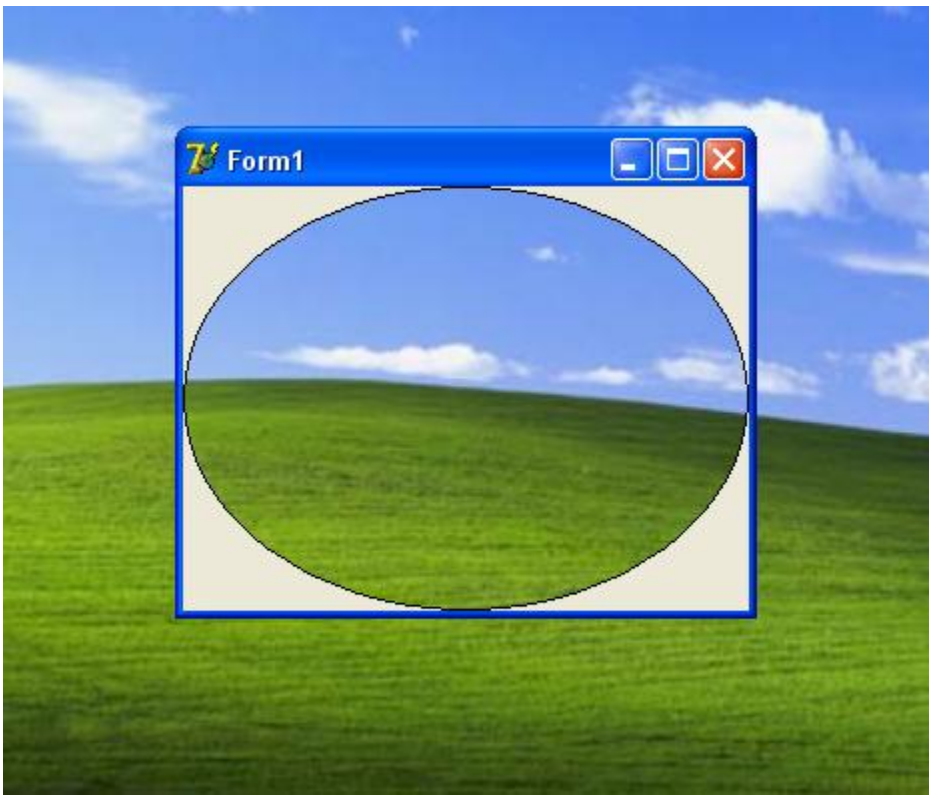


*Рисунок 7.27. Прозрачная форма*

Например, рисунок 7.27 был получен следующим образом. Мы положили на форму компонент **Shape**, превратили его в эллипс (**Shape** = stEllipse), растянули до размеров формы (**Align** = alClient), в форме установили свойство **TransparentColor** в значение True и уравняли в форме значение свойства **TransparentColorValue** со свойством **Brush.Color** компонента **Shape**. После сборки и запуска программы получили «дырявую» форму.

## 7.3.10. Полупрозрачная форма

Форма может быть полупрозрачной (рисунок 7.28). За полупрозрачность формы отвечают свойства **AlphaBlend** и **AlphaBlendValue**. Первое свойство включает и выключает эффект полупрозрачности, а второе определяет силу прозрачности.


*Рисунок 7.28. Полупрозрачная форма*

Например, рисунок 7.28 был получен следующим образом. Мы положили на форму компонент **Image**, загрузили в него картинку (свойство **Picture**), затем в форме установили свойство **AlphaBlend** в значение True и свойство **AlphaBlendValue** — в значение 150. После сборки и запуска программы получили эффект полупрозрачности.

## 7.3.11. Недоступная форма

Иногда бывает нужно просто запретить доступ к форме, не убирая ее с экрана. Для этого служит другое булевское свойство **Enabled**. Обычно оно равно значению True, но стоит ему присвоить противоположное значение, и после запуска приложения вы не сможете сделать форму активной.

Как вы понимаете, все описанные выше свойства доступны не только в окне свойств, но и в редакторе кода, т.е. в тексте программы. При работе с формой на уровне исходного кода вы также получаете доступ к некоторым дополнительным свойствам, которые не видны в окне свойств. Они будут рассматриваться по мере надобности.

## 7.3.12. События формы

Итак, со свойствами мы разобрались и пора сказать пару слов о возникающих при работе с формой событиях. С вашего позволения мы опустим те события формы, которые происходят во всех видимых на экране компонентах (мы о них расскажем позже, когда будем рассматривать компоненты). Перечислим лишь характерные события форм:

- **OnCreate** — происходит сразу после создания формы. Обработчик этого события может установить начальные значения для свойств формы и ее компонентов, запросить у операционной системы необходимые ресурсы, создать служебные объекты, а также выполнить другие действия прежде, чем пользователь начнет работу с формой. Парным для события OnCreate является событие OnDestroy.
- **OnDestroy** — происходит непосредственно перед уничтожением формы. Обработчик этого события может освободить ресурсы, разрушить служебные объекты, а также выполнить другие действия прежде, чем объект формы будет разрушен.

- **OnShow** — происходит непосредственно перед отображением формы на экране. Парным для события OnShow является событие OnHide.
- **OnHide** — происходит непосредственно перед исчезновением формы с экрана. Парным для события OnHide является событие OnShow.
- **OnActivate** — происходит, когда пользователь переключается на форму, т.е. форма становится активной. Парным для события OnActivate является событие OnDeactivate.
- **OnDeactivate** — происходит, когда пользователь переключается на другую форму, т.е. текущая форма становится неактивной. Парным для события OnDeactivate является OnActivate.
- **OnCloseQuery** — происходит при попытке закрыть форму. Запрос на закрытие формы может исходить от пользователя, который нажал на рамке формы кнопку "Закрыть", или от программы, которая вызвала у формы метод Close. Обработчику события OnCloseQuery передается по ссылке булевский параметр CanClose, разрешающий или запрещающий действительное закрытие формы.
- **OnClose** — происходит после события OnCloseQuery, непосредственно перед закрытием формы.
- **OnContextPopup** — происходит при вызове контекстного меню формы.
- **OnMouseDown** — происходит при нажатии пользователем кнопки мыши, когда указатель мыши наведен на форму. После отпускания кнопки мыши в компоненте происходит событие **OnMouseUp**. При перемещении указателя мыши над формой периодически возникает событие **OnMouseMove**, что позволяет отслеживать позицию указателя.
- **OnMouseWheelUp** — происходит, когда колесико мыши проворачивается вперед (от себя).
- **OnMouseWheelDown** — происходит, когда колесико мыши проворачивается назад (на себя).
- **OnMouseWheel** — происходит, когда колесико мыши проворачивается в любую из сторон.
- **OnStartDock** — происходит, когда пользователь начинает буксировать стыкуемый компонент.
- **OnGetSiteInfo** — происходит, когда стыкуемый компонент запрашивает место для стыковки.
- **OnDockOver** — периодически происходит при буксировке стыкуемого компонента над формой.
- **OnDockDrop** — происходит при стыковке компонента (см. главу 10).
- **OnEndDock** — происходит по окончании стыковки компонента.
- **OnUnDock** — происходит, когда пользователь пытается отстыковать компонент.
- **OnDragDrop** — происходит, когда пользователь опускает в форму буксируемый объект.
- **OnDragOver** — периодически происходит при буксировке объекта над формой.
- **OnCanResize** — происходит при попытке изменить размеры формы. Запрос на изменение размеров может исходить от пользователя. Обработчику события OnCanResize передается по ссылке булевский параметр **Resize**, разрешающий или запрещающий действительное изменение размеров формы.
- **OnResize** — происходит при изменении размеров формы.
- **OnConstrainedResize** — происходит при изменении размеров формы и позволяет на лету изменять минимальные и максимальные размеры формы.
- **OnShortCut** — происходит, когда пользователь нажимает клавишу на клавиатуре (до события **OnKeyDown**, см. параграф 7.5.5). Позволяет перехватывать нажатия клавиш еще до того, как они дойдут до стандартного обработчика формы.

## 7.4. Несколько форм в приложении

Часто одной формы для решения задачи бывает мало. Поэтому сейчас мы рассмотрим, как добавить в проект новую форму, выбрать главную форму приложения, переключаться между формами. Затем мы расскажем, как на этапе работы приложения решается вопрос показа форм на экране.

### 7.4.1. Добавление новой формы в проект

Добавить в проект новую форму крайне просто: выберите команду меню **File | New | Form** и на экране сразу появиться вторая форма. При этом в окне редактора кода автоматически появится соответствующий новой форме программный модуль. Только что созданной форме дайте имя **SecondaryForm** (свойство **Name**) и заголовок **Secondary** (свойство **Caption**) — рисунок 7.29.

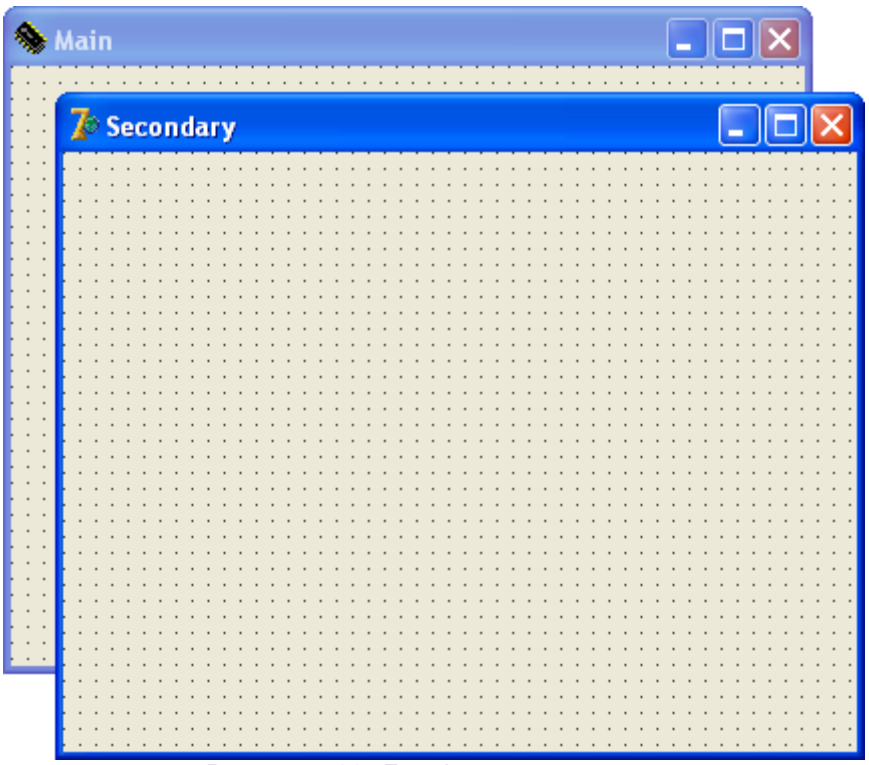


*Рисунок 7.29. Две формы в проекте*

Сохраните модуль с новой формой под именем **Second.pas** — форма нам еще понадобится.

## 7.4.2. Добавление новой формы из Хранилища Объектов

Существует и второй, более продуктивный, способ создания форм. Он основан на использовании готовых форм, существующих в Хранилище Объектов среды Delphi. *Хранилище Объектов* (**Object Repository**) содержит заготовки форм, программных модулей и целых проектов, которые вы можете либо просто скопировать в свой проект, либо унаследовать, либо вообще использовать напрямую. Чтобы взять новую форму из Хранилища объектов, выберите в меню команду **File | New | Other...**. Среда Delphi откроет окно, показанное на рисунке 7.30:

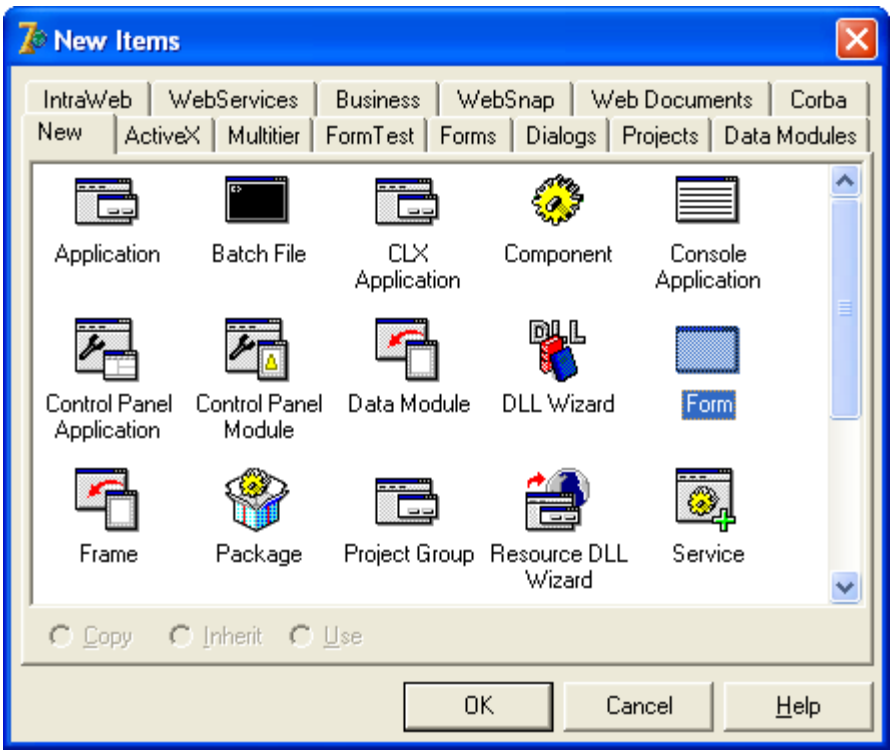


*Рисунок 7.30. Окно создания новой формы или другого элемента проекта*

Если на вкладке **New** диалогового окна выбрать значок с подписью **Form**, то в проект добавится обычная пустая форма, как по команде меню **File | New Form**. Если вас интересуют формы с «начинкой», обратитесь к вкладкам **Forms** и **Dialogs** (рисунок 7.31).

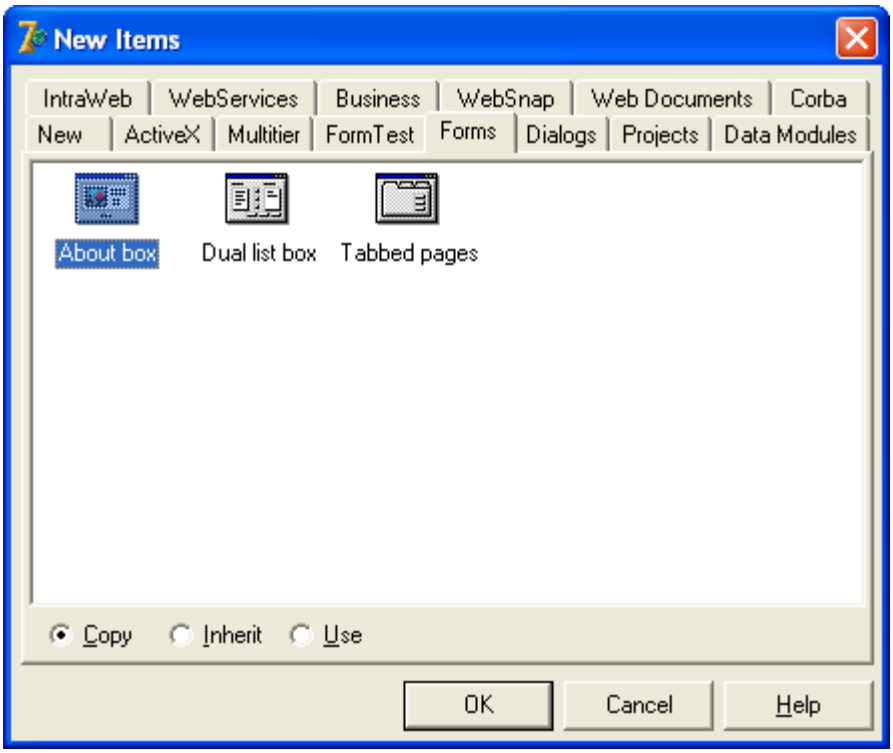


*Рисунок 7.31. Быстрое создание формы с «начинкой»*

На вкладках **Forms** и **Dialogs** существует переключатель, указывающий, что нужно сделать с формой-заготовкой: копировать (**Copy**), наследовать (**Inherit**) или использовать (**Use**). Отличие между ними состоит в следующем:

- **Copy** — означает, что в проект помещается полная копия формы-заготовки.

- **Inherit** — означает, что добавляемая в проект форма создается методом наследования от формы-заготовки, находящейся в Хранилище Объектов;
- **Use** — означает, что в проект добавляется сама форма-заготовка; изменение формы в проекте означает изменение формы-заготовки в Хранилище Объектов.

Какой из режимов использовать — зависит от условия задачи. Режим **Copy** хорош просто тем, что не с нуля начинает разработку новой формы. Режим **Inherit** полезен, когда в проекте существует несколько форм, у которых совпадают некоторые части. В этом случае все похожие между собой формы порождаются от какой-то одной формы, реализующей общую для всех наследников часть. Режим **Use** позволяет подкорректировать форму-заготовку прямо в Хранилище Объектов.

Для нашего учебного примера двух форм достаточно, поэтому вернемся к уже созданным формам, нажав кнопку Cancel.

## 7.4.3. Переключение между формами во время проектирования

Иногда за формами становиться трудно уследить. Навести порядок помогает окно **View Form**, для вызова которого служит команда меню **View | Forms...** (рисунок 7.32).

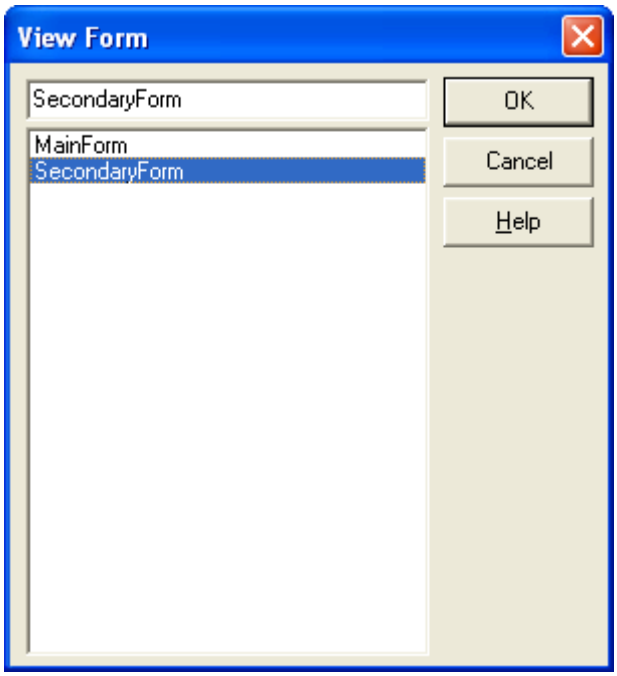


*Рисунок 7.32. Окно для переключения на другую форму*

Выберите в этом окне форму, с которой собираетесь работать, и щелкните по кнопке **OK**. Выбранная форма сразу же станет активной.

## 7.4.4. Выбор главной формы приложения

Когда в проекте несколько форм, возникает вопрос: какая из них главная. Давайте не будем ломать голову, а обратимся к известному вам диалоговому окну **Project Options** и посмотрим на вкладке **Forms**, какая форма выбрана в выпадающем списке **Main form** (рисунок 7.33).

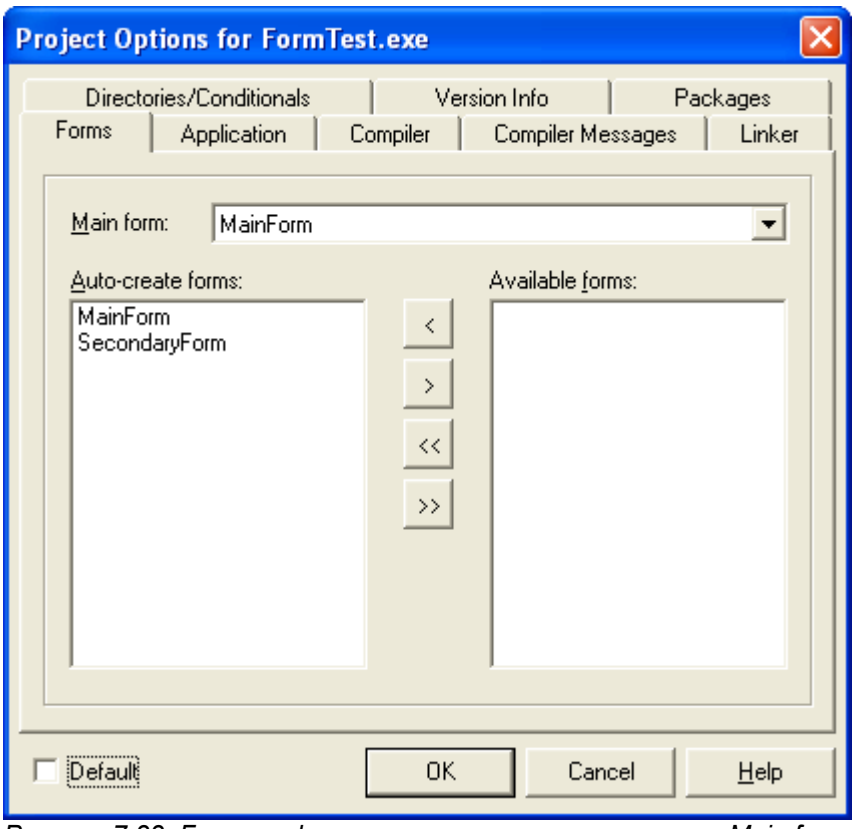


*Рисунок 7.33. Главная форма в проекте записана в поле Main form*

Вы обнаружите, что выбрана форма **MainForm**, которая была добавлена в проект первой (среда Delphi создает ее автоматически при создании нового проекта). Вы можете выбрать другую форму — и тогда она будет отображаться при запуске приложения. В данном случае этого делать не надо, поскольку главная форма уже установлена правильно.

### 7.4.5. Вызов формы из программы

Работая с несколькими формами, вы должны принимать во внимание, что после загрузки приложения отображается только главная форма. Остальные формы хотя и создаются вслед за ней автоматически, на экране сразу не показываются, а ждут пока их вызовут. Форму можно вызвать для работы двумя разными способами:

- вызвать для работы в обычном режиме с помощью метода **Show**. В этом режиме пользователь может работать одновременно с несколькими формами, переключаясь между ними;
- вызвать для работы в монопольном режиме с помощью метода **ShowModal**. В этом режиме пользователь не может переключиться на другую форму, пока не завершит работу с данной формой;

Покажем, как реализуются эти способы на примере вызова формы **SecondaryForm** из формы **MainForm**.

Чтобы форма **SecondaryForm** была доступна для использования формой **MainForm**, необходимо подключить модуль формы **SecondaryForm** к модулю формы **MainForm**. Это делается очень просто.

1. Активизируйте форму **MainForm** и выберите в главном меню команду **File | Use Unit...** . В появившемся диалоговом окне выберите модуль **Second** (так называется модуль формы **SecondaryForm**) и нажмите кнопку OK (рисунок 7.34).

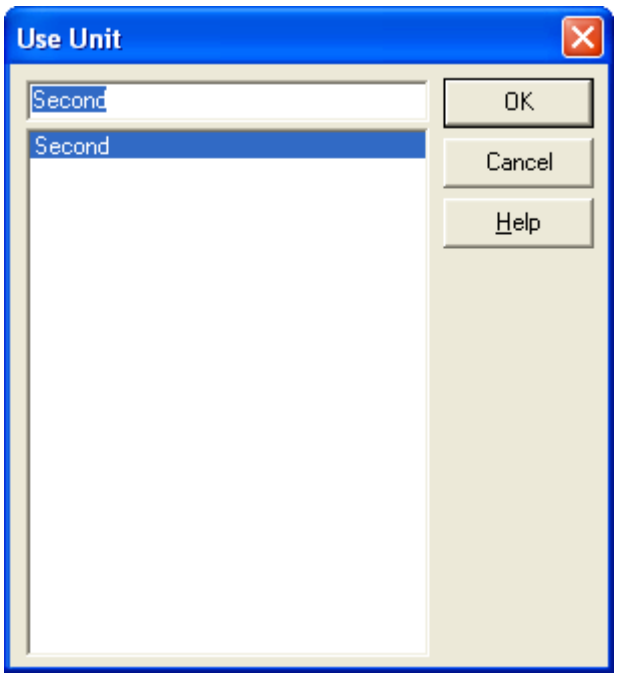


*Рисунок 7.34. Окно для выбора подключаемого модуля*

На экране не произойдет видимых изменений, но в секции **implementation** программного модуля Main добавится строка

```
uses Second;
```

Теперь обеспечим вызов формы **SecondaryForm** из формы **MainForm**. В большинстве случаев формы вызываются при нажатии некоторой кнопки. Добавим такую кнопку на форму **MainForm**.

2. Отыщите в палитре компонентов на вкладке **Standard** значок с подсказкой **Button**, щелкните по нему, а затем щелкните по форме **MainForm**. На форме будет создана кнопка **Button1** и в окне свойств отобразится список ее свойств. Перейдите к свойству **Caption** и замените текст **Button1** на текст **Secondary** (рисунок 7.35).

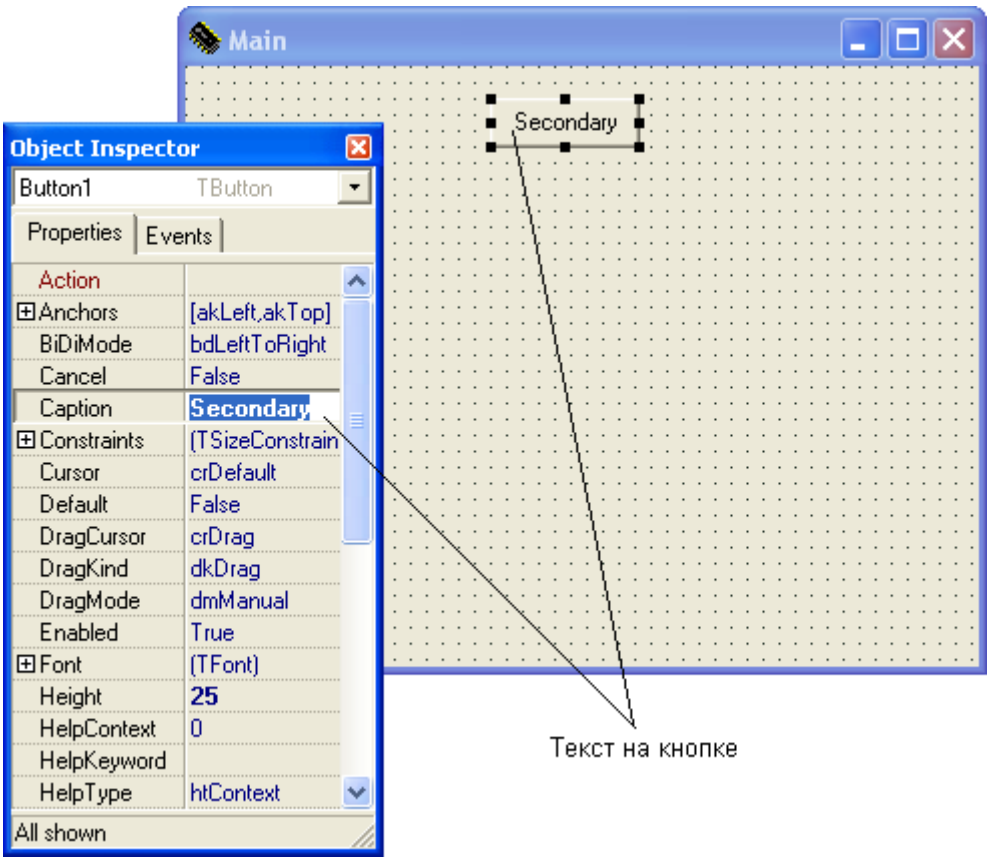


*Рисунок 7.35. Текст на кнопке записывается в свойстве Caption*

Чтобы при нажатии кнопки отображалась форма **SecondaryForm**, необходимо для этой кнопки определить обработчик события **OnClick**. Это делается очень просто.

3. Активизируйте в окне свойств вкладку **Events** и сделайте двойной щелчок в поле события **OnClick**. Среда Delphi определит для кнопки обработчик события, поместив программную заготовку в исходный текст модуля **Main**. Вставьте в тело обработчика оператор **SecondaryForm.Show**, в результате метод обработки события примет следующий вид:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  SecondaryForm.Show;
end;
```

4. Выполните компиляцию и запустите приложение. Когда на экране покажется форма **MainForm**, нажмите кнопку **Secondary**. На экране покажется еще одна форма — **SecondaryForm**. Вы можете произвольно активизировать любую из двух форм (рисунок 7.36).

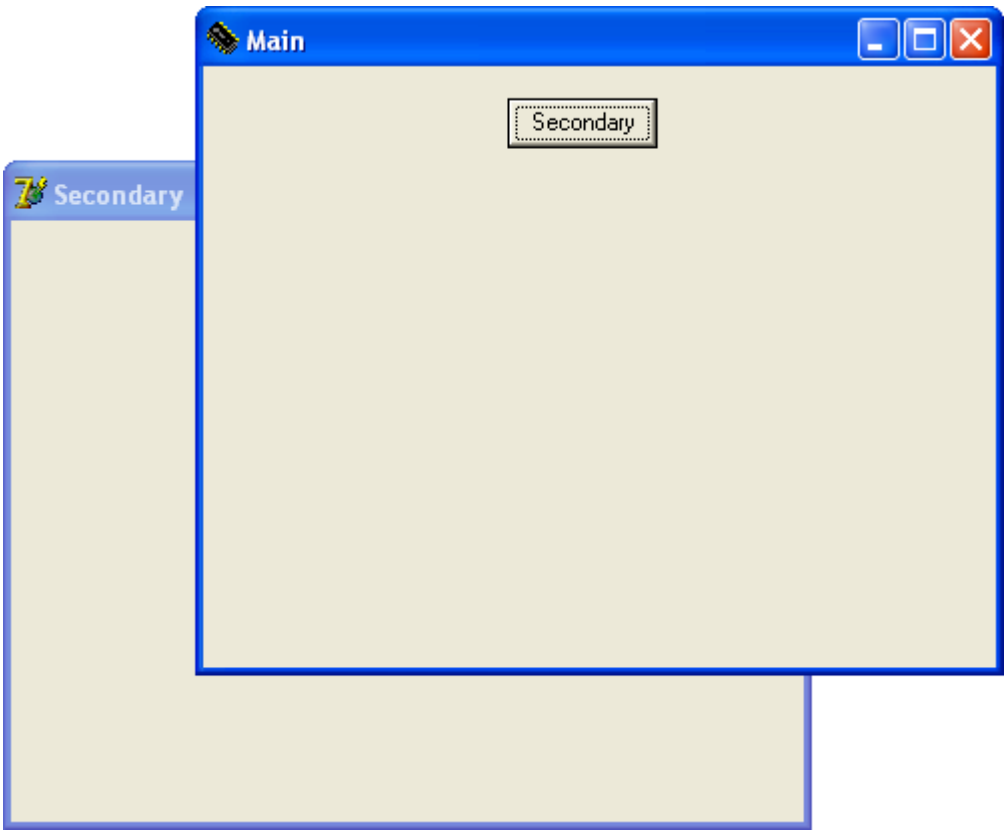


*Рисунок 7.36. Приложение с двумя формами на экране*

Таким образом, при использовании метода **Show** пользователь может работать одновременно с несколькими формами, переключаясь между ними.

Внимание! Переключаясь между формами **MainForm** и **SecondaryForm**, вы можете предположить, что они равноправны. Однако на самом деле это не так. Форма **MainForm** является главной, а форма **SecondaryForm** — второстепенной. В чем это проявляется? Да хотя бы в том, что если вы закроете форму **MainForm**, то форма **SecondaryForm** тоже закроется, и приложение завершится. Если же вы закроете форму **SecondaryForm**, то форма **MainForm** на экране останется.

Ситуация, когда пользователю предлагается для работы сразу несколько доступных форм, встречается редко. Поэтому для показа формы в основном применяется метод **ShowModal**. Он обеспечивает работу формы в монопольном режиме, не возвращая управление до тех пор, пока пользователь не закроет форму.

5. Посмотрим, что произойдет, если в предыдущем примере заменить вызов метода **Show** на **ShowModal**.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  SecondaryForm.ShowModal;
end;
```

6. После компиляции и запуска приложения нажмите на форме **MainForm** кнопку **Secondary**. После появления формы **SecondaryForm** попробуйте активизировать форму **MainForm**. У вас ничего не выйдет, поскольку на этот раз форма **SecondaryForm** отображается в монопольном режиме (рисунок 7.37).

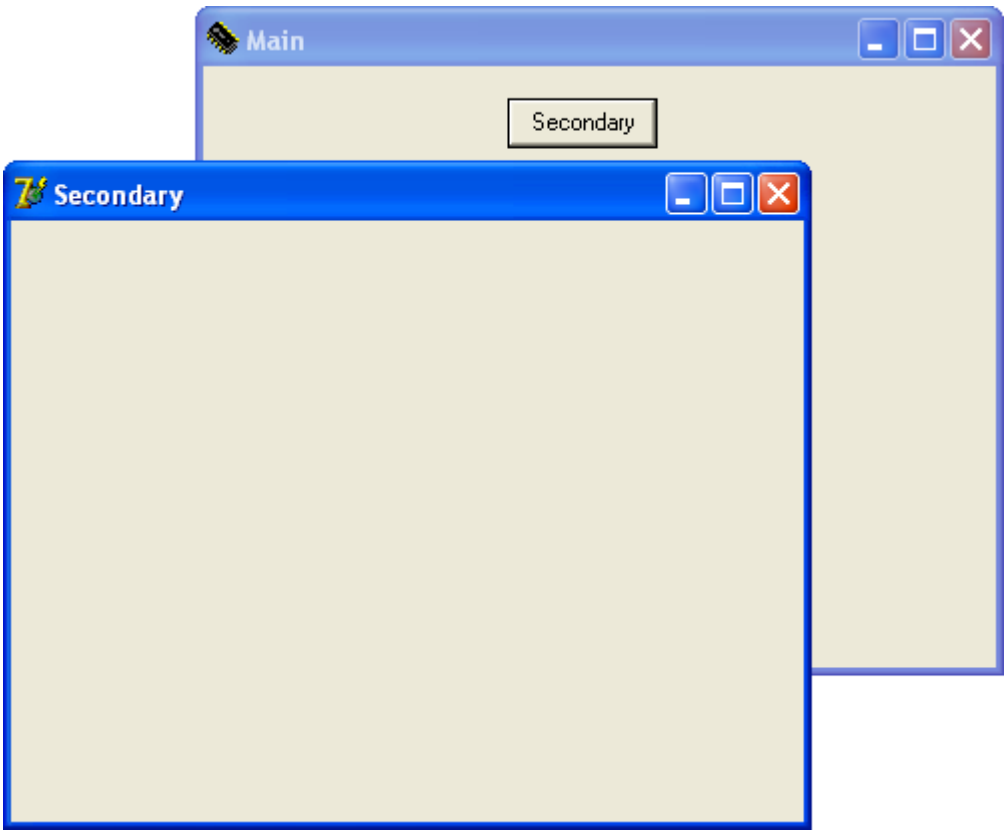


*Рисунок 7.37. Вторая форма работает в монопольном режиме*

Только закрыв форму **SecondaryForm**, вы вернетесь к форме **MainForm**. Теперь понятно и назначение метода **ShowModal**. С его помощью организуется пошаговый (диалоговый) режим взаимодействия с пользователем, который подробно рассмотрен в главе 9.

На этом мы закончим разговор о формах и перейдем к их содержимому — компонентам.

# 7.5. Компоненты

## 7.5.1. Понятие компонента

Понятие компонента является фундаментальным для среды Delphi. Без компонентов все преимущества визуальной разработки приложений исчезают и говорить становится не о чем. Поэтому соберите все силы и внимательно прочитайте этот параграф, пытаясь усвоить не только технику использования компонентов, но и саму их суть.

Существует два взгляда на компоненты.

- Взгляд снаружи, точнее из среды визуальной разработки приложений. С этой точки зрения компоненты — это самодостаточные строительные блоки, которые вы берете из палитры компонентов и переносите на форму для создания собственно приложения. Примеры компонентов вам известны: это кнопки, списки, надписи и др.
- Существует еще и взгляд изнутри, т.е. взгляд из программы на языке Delphi. С этой точки зрения компоненты — это классы, порожденные прямо или косвенно от класса **TComponent** и имеющие **published**-свойства. Экземпляры компонентов — это объекты этих классов, существующие в качестве полей формы. Среди опубликованных свойств компонентов обязательно присутствует имя (**Name**), под которым экземпляр компонента представляется в окне свойств.

Объединение этих двух точек зрения дает цельное представление о том, что такое компоненты. При работе с компонентами из среды визуальной разработки приложений вы всегда видите их лицевую сторону. Однако как только вы начинаете писать обработчики событий, и управлять компонентами программно, вы соприкасаетесь с программной стороной компонентов, суть

которой — объекты. Таким образом, среда Delphi обеспечивает симбиоз визуального и объектно-ориентированного программирования.

При анализе структуры компонента обнаруживается, что его природа троична и лучше всего описывается формулой:

Компонент = состояние (свойства) + поведение (методы) + обратная реакция (события).

- *Состояние компонента* определяется его свойствами. Свойства бывают изменяемые (для чтения и записи) и неизменяемые (только для чтения). Помимо этого, свойства могут получать значения либо на этапе проектирования (design-time), либо только во время выполнения программы (run-time). Первые устанавливаются для каждого компонента в окне свойств и определяют начальное состояние компонента. Во время выполнения приложения эти свойства могут быть изменены программно, соответственно изменится внешний вид и поведение компонента. Вторая группа — это свойства, которые не видны в окне свойств, и управлять которыми можно только программно. С точки зрения языка Delphi различие между этими группами свойств состоит в том, что первые объявлены в секции **published**, а вторые — в секции **public**.
- *Поведение компонента* описывается с помощью его процедур и функций (*методов*). Вызовы методов компонента помещаются в исходный код программы и происходят только во время выполнения приложения. Методы не имеют под собой визуальной основы.
- *Обратная реакция компонента* — это его *события*. События позволяют, например, связать нажатие кнопки с вызовом метода формы. События реализуются с помощью свойств, содержащих указатели на методы (см. гл. 3).

## 7.5.2. Визуальные и невизуальные компоненты

Все компоненты делятся на две группы: визуальные и невизуальные компоненты (рисунок 7.38).

- *Визуальные компоненты* (visual components) — это видимые элементы пользовательского интерфейса: кнопки, метки, блоки списков и др. Они выглядят одинаково и на стадии проектирования, и во время работы приложения.
- *Невизуальные компоненты* (non-visual components) — это, так сказать, бойцы невидимого фронта; они работают, но сами на экране не видны. К невизуальным компонентам относятся таймер, компоненты доступа к базам данным и др. В процессе проектирования такие компоненты представляются на форме небольшим значком. Их свойства устанавливаются в уже известном вам окне свойств. Некоторые компоненты хоть и являются невизуальными, могут что-нибудь отображать на экране. Например, невизуальный компонент **MainMenu** отображает на форме полосу главного меню, а компонент **OpenDialog** — стандартное диалоговое окно выбора файла.

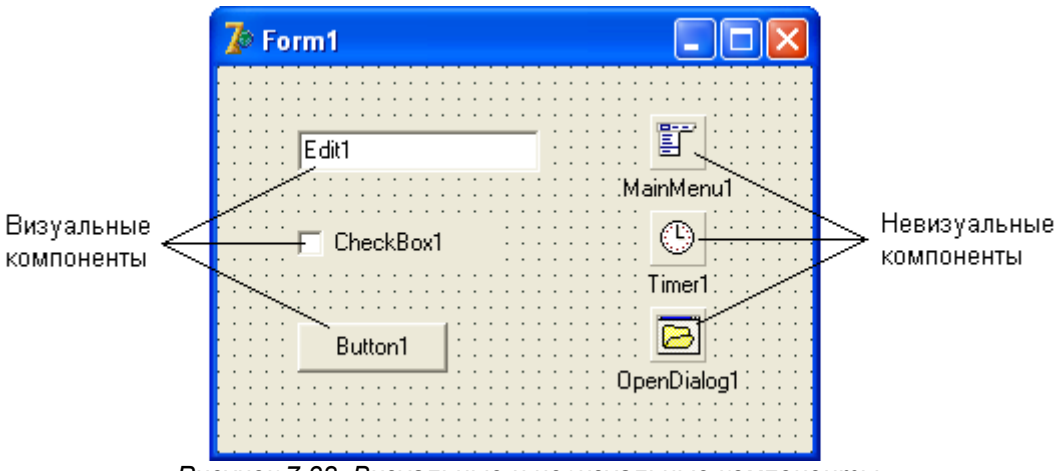


*Рисунок 7.38. Визуальные и невизуальные компоненты*

Невизуальные компоненты могут иметь подписи (рисунок 7.38). Отображение подписей обеспечивается установкой переключателя **Show component captions** в окне **Environment Options** на вкладке **Designer**. Окно вызывается по команде меню **Tools | Environment Options...**

## 7.5.3. «Оконные» и «графические» компоненты

Визуальные компоненты подразделяются на компоненты, рисуемые оконной системой Windows, и компоненты, рисуемые графической библиотекой VCL (рисунок 7.39). На программистском жаргоне первые называют «оконными» компонентами, а вторые — «графическими» компонентами.

- *«Оконные» компоненты* (windowed controls) являются специализированными окнами внутри окна формы. Их самое главное качество — способность получать фокус ввода. К числу оконных компонентов относятся, например, компоненты **Button**, **RadioButton**, **CheckBox**, **GroupBox**, и т.д. Некоторые оконные компоненты (**GroupBox**, **TabControl**, **PageControl**) способны содержать другие визуальные компоненты и называются *контейнерами* (container controls). Отображение оконных компонентов обеспечивается операционной системой Windows. Для профессионалов, имевших дело Windows API, заметим, что оконные компоненты имеют свойство **Handle**. Оно связывает компонент среды Delphi с соответствующим объектом операционной системы.
- *«Графические» компоненты* (graphical controls) не являются окнами, поэтому не могут получать фокус ввода и содержать другие визуальные компоненты. Графические компоненты не основаны на объектах операционной системы Windows, их отображение полностью выполняет библиотека VCL. К числу графических компонентов относятся, например, компоненты **SpeedButton**, **Image**, **Bevel** и т.д.

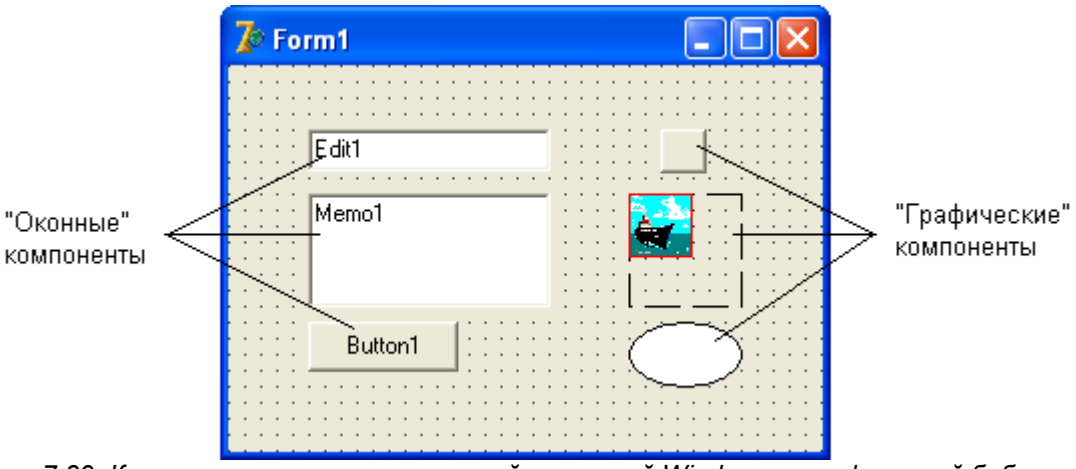


*Рисунок 7.39. Компоненты, рисуемые оконной системой Windows и графической библиотекой Delphi*

Общая классификация компонентов составлена, поэтому перейдем к обсуждению их свойств и событий. Очевидно, каждый компонент обладает специфичным набором свойств и событий и, казалось бы, изучать их следует в контексте изучения компонента. Так мы и будем поступать в будущем при рассмотрении отличительных свойств компонентов. Однако сейчас имеет смысл рассмотреть общие для большинства компонентов свойства и события.

Невизуальные компоненты практически не имеют общих свойств и событий, единственные общие для них свойства — это **Name** (комментариев не требует) и **Tag** (целочисленное значение, не несущее смысловой нагрузки — вы можете использовать его по своему усмотрению). А вот визуальные компоненты имеют много общих свойств и событий, которые мы сейчас и рассмотрим.

## 7.5.4. Общие свойства визуальных компонентов

Визуальные компоненты имеют ряд общих свойств:

- **Left** и **Top** — местоположение визуального компонента внутри формы (или внутри компонента-владельца).
- **Width** и **Height** — горизонтальный и вертикальный размеры компонента соответственно.
- **Anchors** — позволяет привязать границы компонента к границам формы. Привязанная граница компонента будет следовать за соответствующей границей формы при изменении размеров формы. Поэкспериментируйте со значениями этого свойства и вы быстро уловите логику его работы.

- **BiDiMode** — позволяет сделать так, чтобы текст читался справа налево (используется при работе с восточными языками). Компонент либо использует свое собственное значение свойства, либо копирует его из компонента-владельца, если вспомогательное свойство **ParentBiDiMode** равно значению True.
- **Caption** — надпись компонента. Установленная в свойстве текстовая строка может содержать специальный символ '&' (амперсант). Если в строке встречается амперсант, то следующий за ним символ отображается подчеркнутым (амперсант при этом не отображается). Нажатие соответствующей символьной клавиши на клавиатуре в сочетании с клавишей Alt активизирует компонент.
- **Constraints** — ограничения на размеры компонента. Вложенные свойства **MinWidth** и **MinHeight** определяют минимальные ширину и высоту, а вложенные свойства **MaxWidth** и **MaxHeight** — максимальные ширину и высоту соответственно.
- **Color** — цвет компонента. Компонент либо использует свой собственный цвет, либо копирует цвет содержащего компонента. Это определяется значением свойства **ParentColor**. Если свойство ParentColor имеет значение True, то изменение цвета у содержащего компонента (например, формы) автоматически приводит к изменению цвета вложенного компонента (например, кнопки). Однако, если вы измените значение свойства Color, то свойство ParentColor автоматически примет значение False, и компонент получит свой собственный цвет.
- **Cursor** — определяет, какой вид принимает указатель мыши, когда пользователь наводит его на компонент. Каждому варианту указателя соответствует своя целочисленная константа (например, константа **crArrow** соответствует обычному указателю в виде стрелки). Полный список значений с описанием вы сможете найти в справочной системе среды Delphi.
- **DragCursor** — вид указателя мыши, когда пользователь буксирует объект над компонентом. Этот вид курсора устанавливается лишь в том случае, если объект может быть принят (см. главу 10).
- **DragKind** — определяет поведение компонента при буксировке: просто буксировка (**dkDrag**) или стыковка (**dkDock**). В зависимости от значения этого свойства возникает та или иная цепочка событий: цепочка событий буксировки или цепочка событий стыковки.
- **DragMode** — определяет режим буксировки компонента по экрану. Если в свойстве установлено значение dmManual (принято по умолчанию), то буксировка должна инициироваться программно. Если же в свойстве установлено значение dmAutomatic, то компонент уже готов к буксировке, пользователю достаточно навести указатель мыши на компонент, нажать кнопку мыши и, удерживая ее, отбуксировать компонент в нужное место.
- **Enabled** — определяет, доступен ли компонент для пользователя. Если свойство имеет значение True, то компонент доступен, а если значение False, то недоступен. Недоступный компонент обычно имеет блеклый вид.
- **Font** — шрифт надписи на компоненте. Параметры шрифта задаются с помощью вложенных свойств **CharSet**, **Color**, **Name**, **Size**, **Style**, **Height**, **Pitch**, **Weight**. Компонент либо использует свой собственный шрифт, либо копирует шрифт содержащего компонента. Это определяется значением свойства **ParentFont**. Если свойство ParentFont имеет значение True, то изменение шрифта у содержащего компонента (например, формы) автоматически приводит к изменению шрифта у вложенного компонента (например, кнопки). Однако, если вы измените значение свойства Font, то свойство ParentFont автоматически примет значение False, и компонент получит свой собственный шрифт для надписи.
- **HelpType** — определяет, каким образом в файле справки будет осуществляться поиск темы, соответствующей компоненту. Когда компонент обладает фокусом ввода, пользователь может нажать клавишу F1, чтобы получить оперативную справку. Поиск соответствующей темы осуществляется либо по номеру, заданному в свойстве **HelpContext**, либо по ключевому слову, заданному в свойстве **HelpKeyword**. В первом случае свойство HelpType должно иметь значение **htContext**, а во втором — **htKeyword**.
- **HelpContext** — содержит номер соответствующей темы в файле справки. Используется, когда свойство **HelpType** имеет значение **htContext**. Если свойство HelpContext имеет значение 0, то номер темы берется из аналогичного свойства компонента-владельца (как правило, формы).
- **HelpKeyword** — содержит ключевое слово для поиска темы в файле справки. Используется, когда свойство **HelpType** имеет значение **htKeyword**. Если свойство HelpKeyword имеет пустое значение, то поиск осуществляется по ключевому слову, которое берется из аналогичного свойства компонента-владельца (как правило, формы).
- **Hint** — подсказка, появляющаяся над компонентом, когда пользователь временно задерживает над ним указатель мыши. Появление подсказки может быть разрешено или запрещено с помощью свойства **ShowHint**. Значение свойства ShowHint может копироваться из содержащего компонента в зависимости от значения свойства

**ParentShowHint**. Если свойство ParentShowHint имеет значение True, то запрет подсказки для содержащего компонента (например, для формы), автоматически приводит к запрету подсказки для вложенного компонента (например, для кнопки). Однако, если вы измените значение свойства ShowHint, то свойство ParentShowHint автоматически примет значение False, и управление запретом подсказки перейдет к компоненту.

- **PopupMenu** — используется для привязки контекстного меню к компоненту. Это меню вызывается щелчком правой кнопки мыши по компоненту. Меню подробно рассмотрены в главе 8.
- **TabOrder** — содержит порядковый номер компонента в пределах своего компонента-владельца. Это номер очереди, в которой компонент получает фокус ввода при нажатии клавиши Tab на клавиатуре. Свойство TabOrder присутствует только в оконных компонентах.
- **TabStop** — определяет, может ли компонент получать фокус ввода. Если свойство имеет значение True, то компонент находится в очереди на фокус ввода, а если значение False, то — нет. Свойство TabStop присутствует только в оконных компонентах.
- **Visible** — определяет видимость компонента на экране. Если свойство имеет значение True, то компонент виден, а если значение False, то — не виден.

### 7.5.5. Общие события визуальных компонентов

Визуальные компоненты имеют ряд общих событий:

- **OnClick** — происходит в результате щелчка мыши по компоненту.
- **OnContextPopup** — происходит при вызове контекстного меню компонента.
- **OnDblClick** — происходит в результате двойного щелчка мыши по компоненту.
- **OnEnter** — происходит при получении компонентом фокуса ввода. Когда компонент теряет фокус ввода, происходит событие **OnExit**. События OnEnter и OnExit не происходят при переключении между формами и приложениями.
- **OnKeyDown** — происходит при нажатии пользователем любой клавиши (если компонент обладает фокусом ввода). При отпускании нажатой клавиши происходит событие **OnKeyUp**. Если пользователь нажал символьную клавишу, то вслед за событием OnKeyDown и до события OnKeyUp происходит событие **OnKeyPress**. События о нажатии клавиш обычно приходят активному компоненту, обладающему фокусом ввода. Однако с помощью свойства формы **KeyPreview** можно сделать так, чтобы форма перехватывала клавиатурные события до того, как их получит активный компонент. Для этого свойство KeyPreview устананавливается в значение True.
- **OnMouseDown** — происходит при нажатии пользователем кнопки мыши, когда указатель мыши наведен на компонент. После отпускания кнопки мыши в компоненте происходит событие **OnMouseUp**. При перемещении указателя мыши над компонентом, в последнем периодически возникает событие **OnMouseMove**, что позволяет отслеживать позицию указателя.

Для организации буксировки и стыковки, в визуальных компонентах существует еще несколько событий:

- **OnStartDrag** — происходит, когда пользователь начинает что-нибудь буксировать.
- **OnDragOver** — периодически происходит, когда пользователь буксирует что-нибудь над компонентом.
- **OnDragDrop** — происходит, когда пользователь опускает буксируемый объект на компонент.
- **OnEndDrag** — происходит по окончании буксировки объекта.
- **OnStartDock** — происходит, когда пользователь начинает буксировать стыкуемый компонент.
- **OnEndDock** — происходит по окончании стыковки компонента.

Подробно события буксировки и стыковки рассмотрены в главе 10.

## 7.6. Управление компонентами при проектировании

### 7.6.1. Помещение компонентов на форму и их удаление

Чтобы поместить на форму нужный компонент из палитры компонентов, выполните следующие действия:

1. Наведите указатель мыши на значок нужного компонента в палитре и щелкните левой кнопкой мыши.
2. Наведите указатель мыши на нужное место формы и еще раз щелкните левой кнопкой мыши.

Выбранный компонент окажется на форме и будет готов к настройке в окне свойств.

Часто требуется разместить на форме несколько компонентов одного и того же типа, например, кнопок. В этом случае действуйте следующим образом:

1. Наведите указатель мыши на значок нужного компонента в палитре. Нажмите клавишу Shift и, удерживая ее, щелкните левой кнопкой мыши. Отпустите клавишу Shift.
2. Наведите указатель мыши на то место формы, где будет находиться первый компонент, и щелкните левой кнопкой мыши.
3. Наведите указатель мыши на то место формы, где будет размещен следующий компонент, и снова щелкните левой кнопкой мыши. Повторите это действие столько раз, сколько вам нужно компонентов;
4. Разместив последний компонент, наведите указатель мыши на кнопку с изображением стрелки (она расположена в левой части палитры компонентов), и щелкните левой кнопкой мыши. Это будет сигналом, что размещение однотипных компонентов закончено.

Если вы по каким-либо причинам решили убрать лишний компонент с формы, просто выберите его с помощью мыши и нажмите клавишу Del.

## 7.6.2. Выделение компонентов на форме

На стадии проектирования любой компонент может быть выделен на форме. Свойства выделенного компонента видны в окне свойств и доступны для редактирования. Чтобы выделить компонент, достаточно навести на него указатель и нажать кнопку мыши. Вокруг компонента тут же появятся так называемые "точки растяжки" (sizing handles) для изменения размеров компонента по ширине и высоте (рисунок 7.40).

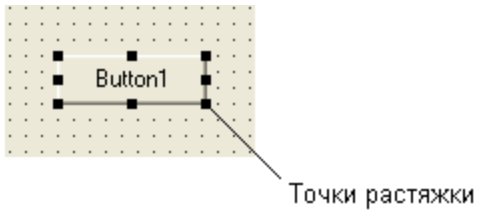


*Рисунок 7.40. Точки растяжки компонента*

При проектировании сложных форм вы столкнетесь с ситуацией, когда сразу в нескольких компонентах нужно установить некоторое свойство в одно и то же значение. Например, в нескольких кнопках установить свойство **Enabled** в значение False. Быстрее всего это можно сделать, если выделить несколько компонентов, после чего перейти к окну свойств и изменить нужное свойство. Когда на форме выделено несколько компонентов, в окне свойств видны только их общие свойства.

Выделить несколько компонентов можно двумя способами:

- Нажать клавишу Shift и, удерживая ее, отметить щелчками мыши все требуемые компоненты, после чего клавишу Shift отпустить. В углах каждого выделенного компонента появятся небольшие квадратики-маркеры.
- Нажать кнопку мыши, когда курсор находится вне компонентов. Затем, удерживая кнопку в нажатом состоянии, протянуть курсор над выделяемыми компонентами, включив их в пунктирный прямоугольник. Когда в пунктирный прямоугольник попадут все требуемые компоненты, кнопку мыши следует отпустить. (Если выделяемые компоненты находятся внутри компонента Panel или GroupBox, то эту операцию нужно выполнять с нажатой клавишей Ctrl.) В результате перечисленных действий в углах всех компонентов, хотя бы

частично попавших в пунктирный прямоугольник, появятся небольшие квадратики-маркеры, свидетельствующие о том, что компоненты выделены.

Вы можете комбинировать оба способа для выделения лишь тех компонентов, которые вам нужны.

Когда на форме выделено несколько компонентов, в окне свойств отображаются лишь их общие свойства. Активизируйте нужное свойство и установите в нем нужное значение. Вы увидите, что эта установка отразится на всех выделенных компонентах (рисунок 7.41).

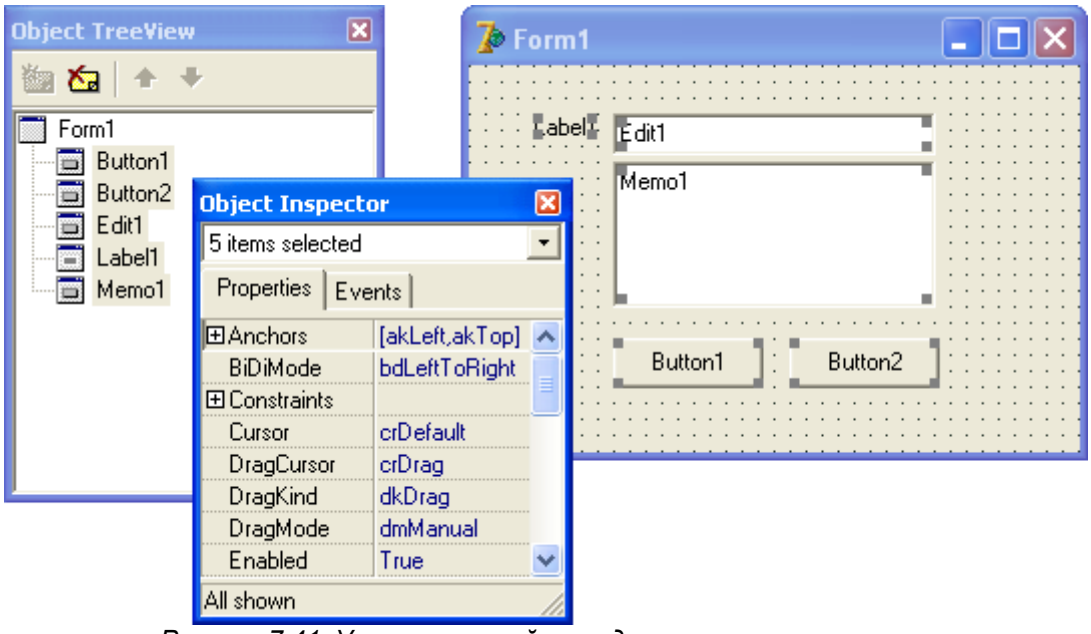


*Рисунок 7.41. Установка свойства для группы компонентов*

Когда на форме выделено несколько компонентов, некоторые свойства могут как бы не иметь значения в окне **Object Inspector** (в поле значения — пусто). Это говорит о том, что свойства имеют различные значения в выделенных компонентах.

## 7.6.3. Перемещение и изменение размеров компонента

Когда все компоненты помещены на форму, нужно оценить ее с точки зрения эстетики. Отойдите от компьютера и посмотрите на форму со стороны. Как правило, в пейзаже обнаружатся некоторые изъяны. Наверняка что-то захочется уменьшить, что-то увеличить, что-то переместить на другое место. Сделать все это — проще простого.

Для перемещения компонента на другое место:

1. Поместите курсор над компонентом, который хотите переместить, и щелкните мышью (компонент тут же окажется в фокусе). Не отпуская кнопки мыши, отбуксируйте компонент на новое место.
2. Когда компонент будет там, где надо, отпустите кнопку мыши.

Изменить размер компонента тоже просто:

1. Щелчком мыши активизируйте компонент, размер которого хотите изменить (он тут же окажется в фокусе);
2. Наведите указатель мыши на точку вертикальной или горизонтальной “растяжки”, при этом вид указателя изменится на двунаправленную стрелку. Нажмите кнопку мыши и, удерживая ее, перемещайте указатель в сторону уменьшения или увеличения размера компонента;
3. Добившись желаемого размера, отпустите кнопку мыши и отведите указатель от точки растяжки (при этом указатель примет обычный вид). Компонент с новыми размерами готов к работе.

Чтобы упростить вам позиционирование и изменение размеров компонентов, форма отображает на этапе разработки *сетку* (grid). Компоненты автоматически выравниваются на пересечении воображаемых линий сетки, когда вы переносите их из палитры компонентов на форму. Изначально шаг между горизонтальными и вертикальными линиями сетки равен 8, но его легко изменить. Для этого выполните команду меню **Tools | Environment Options...** . На экране появится диалоговый блок **Environment Options**. Отыщите поля **Grid size X** и **Grid size Y** на вкладке **Designer** и установите те параметры сетки, которые вам нужны (рисунок 7.42). Кстати, с помощью соседних переключателей можно указать среде Delphi, следует ли вообще показывать сетку (**Display grid**) и следует ли выравнивать по ней компоненты (**Snap to grid**).

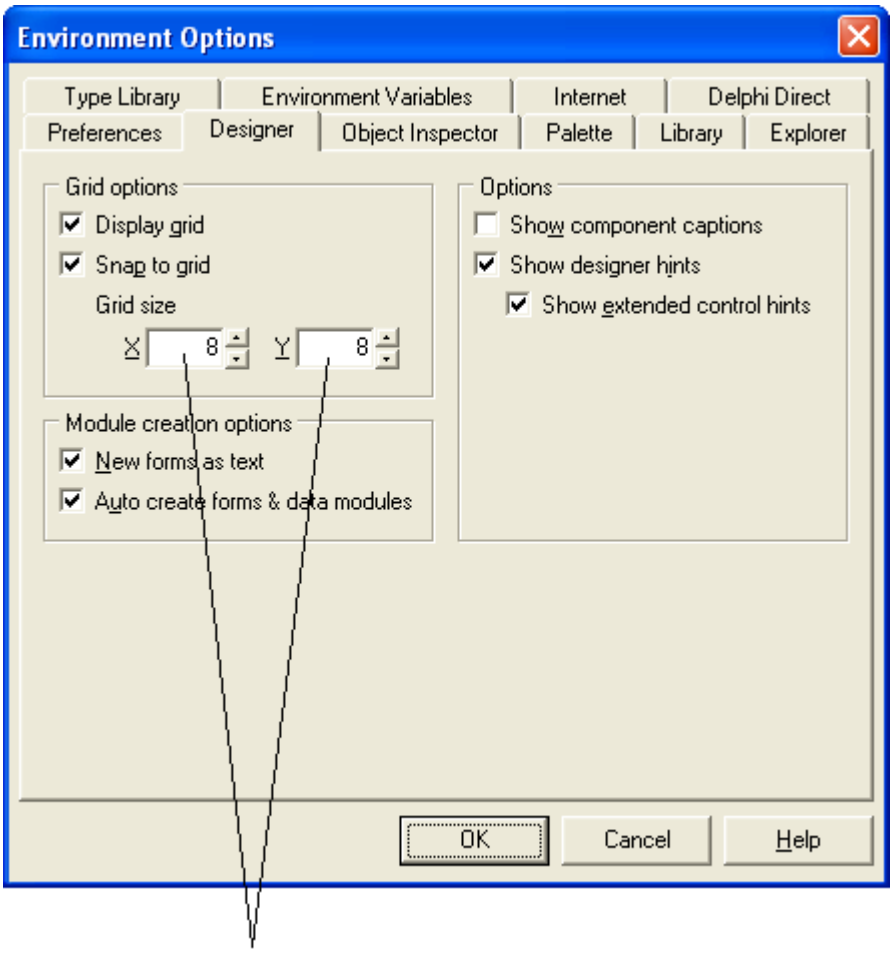


*Рисунок 7.42. Расстояние между узлами сетки по горизонтали и вертикали*

Иногда после грубого размещения компонента на сетке необходимо подогнать его положение и размеры с точностью до точки экрана. Для этого не требуется отключать сетку или изменять ее шаг. Просто выберите компонент на форме и действуйте следующим образом:

- Нажмите клавишу Ctrl и, удерживая ее нажатой, с помощью клавиш со стрелками подвиньте компонент на нужное количество точек экрана. Отпустите клавишу Ctrl.
- Нажмите клавишу Shift и, удерживая ее нажатой, с помощью клавиш со стрелками растяните или сожмите компонент на нужное количество точек экрана. Отпустите клавишу Shift.

## 7.6.4. Выравнивание компонентов на форме

Когда на форме много компонентов, ручное выравнивание становится весьма утомительным занятием. На этот случай в среде Delphi предусмотрены средства автоматизированного выравнивания компонентов. Алгоритм выравнивания следующий:

1. Выделите компоненты, которые собираетесь выравнять. Во всех четырех углах каждого выделенного компонента появятся небольшие квадратики-маркеры;

2. Обратитесь к главному меню и вызовите окно **Alignment** (рисунок 7.43) с помощью команды меню **Edit | Align...** .

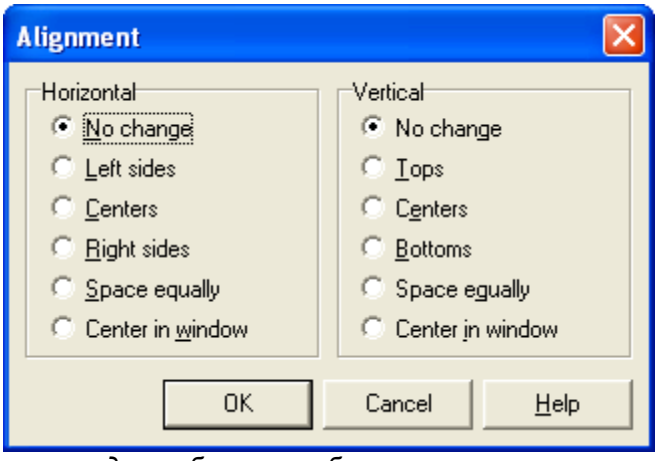


*Рисунок 7.43. Диалоговое окно для выбора способа выравнивания группы компонентов на форме*

3. Выберите в списке то, что вам надо, и нажмите кнопку **OK**. Окно закроется и все компоненты будут выравнены согласно вашим указаниям.

Если компонентов на форме много и вам предстоит большая работа по их выравниванию, откройте окно **Align** (с помощью команды меню **View | Alignment Palette**) и используйте его на втором шаге приведенного выше алгоритма (рисунок 7.44).

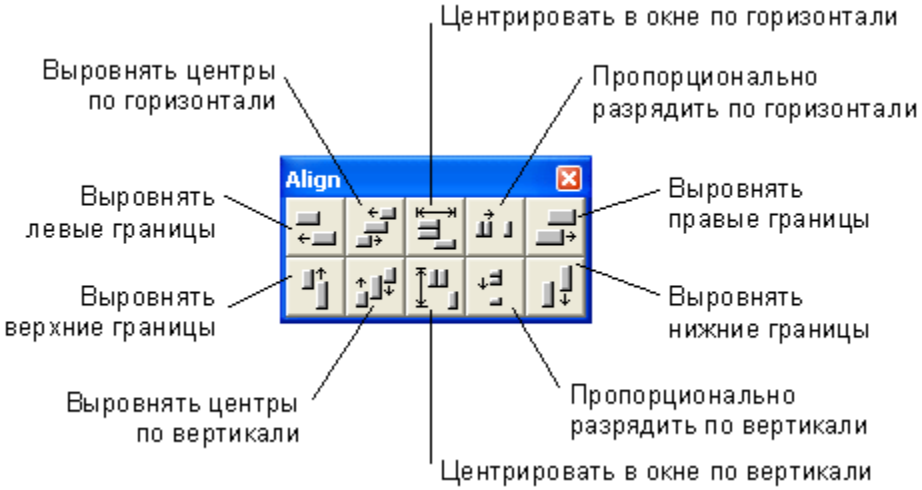


*Рисунок 7.44. Вспомогательная панель кнопок для выравнивания группы компонентов на форме*

## 7.6.5. Использование Буфера обмена

При работе с несколькими формами иногда встает задача копирования и перемещения компонентов с одной формы на другую. Обычное перетягивание здесь не помогает. Проблема легко решается с помощью Буфера обмена:

1. С помощью известных приемов выберите на первой форме компоненты, которые вы желаете скопировать или переместить на вторую форму.
2. Если вы собираетесь скопировать компоненты, выберите в меню команду **Edit | Copy**. Если вы собираетесь переместить компоненты, выберите в меню команду **Edit | Cut**. Компоненты окажутся в Буфере обмена.
3. Активизируйте вторую форму и выберите в меню команду **Edit | Paste**. По этой команде среда Delphi извлечет компоненты из Буфера обмена и поместит их на активную форму.

Добавим, что команды работы с Буфером обмена применяются не только для копирования и перемещения компонентов с одной формы на другую, но также для копирования и перемещения компонентов в пределах одной формы между разными компонентами-владельцами, например для

перемещения кнопок с одной панели на другую. Необходимость использования Буфера обмена в этом случае вызвана тем, что компоненты твердо знают своего владельца (например, кнопки знают панель, на которой они расположены), поэтому обычная операция буксировки ни к чему не приводит.

Итак, вы уже много знаете о компонентах, и дальше углубляться в них не имеет смысла. Начиная со следующей главы, мы начнем знакомить вас с элементами пользовательского интерфейса: меню, панелью инструментов, строкой состояния, диалоговыми окнами и др. — вот там и поговорим о деталях. А сейчас скажем несколько слов о тех объектах, которые усердно работают "за кулисами" приложения и обеспечивают ему доступ к различным ресурсам компьютера, например экрану, принтеру, Буферу обмена и др.

# 7.7. Закулисные объекты приложения

## 7.7.1. Application — главный объект, управляющий приложением

То, о чем мы рассказали выше — это внешняя сторона приложения. А что же происходит внутри? Дело обстоит так. Над всеми формами и компонентами стоит объект **Application** (класса TApplication), олицетворяющий собой приложение в целом. Это главное "действующее лицо", которое создается в начале выполнения любого приложения. Объект **Application** держит в руках все нити управления: создает главную и второстепенные формы, уничтожает их, обслуживает исключительные ситуации. Вы, кстати, уже встречались с ним в файле проекта:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Объект **Application** отсутствует в палитре компонентов, поэтому его свойства можно изменять только из программы. Кратко рассмотрим наиболее важные свойства этого объекта:

- **Active** — равно значению True, если приложение активно.
- **AutoDragDocking** — режим автоматической или ручной стыковки форм и компонентов. В автоматическом режиме (значение True) стыковка происходит по окончании буксировки при отпускании кнопки мыши. В ручном режиме (значение False) для стыковки необходимо удерживать клавишу Ctrl при отпускании кнопки мыши.
- **BiDiKeyboard** — раскладка клавиатуры при работе с восточными языками.
- **BiDiMode** — позволяет сделать так, чтобы надписи читались справа налево (используется при работе с восточными языками).
- **CurrentHelpFile** — имя файла справки активной формы программы (каждая форма может иметь свой собственный файл справки). Если у активной формы нет своего файла справки, то в свойстве **CurrentHelpFile** просто дублируется значение свойства **HelpFile**.
- **HintColor** — цвет фона всплывающей подсказки.
- **HintHidePause** — время, в течение которого всплывающая подсказка задерживается на экране.
- **HintPause** — задержка перед появлением всплывающей подсказки.
- **HintShortCuts** — определяет, включается ли в текст подсказки название комбинации клавиш.
- **HintShortPause** — время, через которое появляется всплывающая подсказка, если в данный момент на экране уже отображена другая подсказка.
- **MainForm** — указывает главную форму приложения. По умолчанию главной считается первая создаваемая форма.
- **NonBiDiKeyboard** — раскладка клавиатуры.

- **ExeName** — содержит полное имя (включая маршрут) выполняемого файла программы. Имя выполняемого файла совпадает с именем главного файла проекта. Если имя проекта не было указано, то по умолчанию выполняемому файлу назначается имя Project1.
- **Title** — содержит название приложения, которое отображается на Панели Задач во время работы. По умолчанию значением свойства является имя главного файла проекта.
- **HelpFile** — содержит имя файла справочника, который используется программой для отображения оперативной справочной информации по формам и компонентам.
- **HelpSystem** — интерфейс к справочной системе.
- **Icon** — содержит значок, отображаемый на Панели Задач во время работы программы. Значок отображается слева от названия (см. **Title**).
- **UpdateFormatSettings** — включает автоматическое обновление форматных строк в программе вслед за изменением этих параметров в операционной системе. Форматные строки управляют показом даты, времени, денежных единиц и др.
- **UpdateMetricSettings** — включает автоматическое обновление шрифта и фона системных надписей (всплывающих подсказок и подписей значков) при изменении настроек экрана в операционной системе.
- **Terminated** — значение True говорит о том, что программа находится в процессе завершения.

Если вы желаете задать заголовок (свойство **Title**), значок (свойство **Icon**) и имя файла справочника (свойство **HelpFile**) для приложения, не корректируйте главный программный файл, а обратитесь лучше к диалоговому окну **Project Options** (рисунок 7.45), которое появляется по команде меню **Project | Options...** .

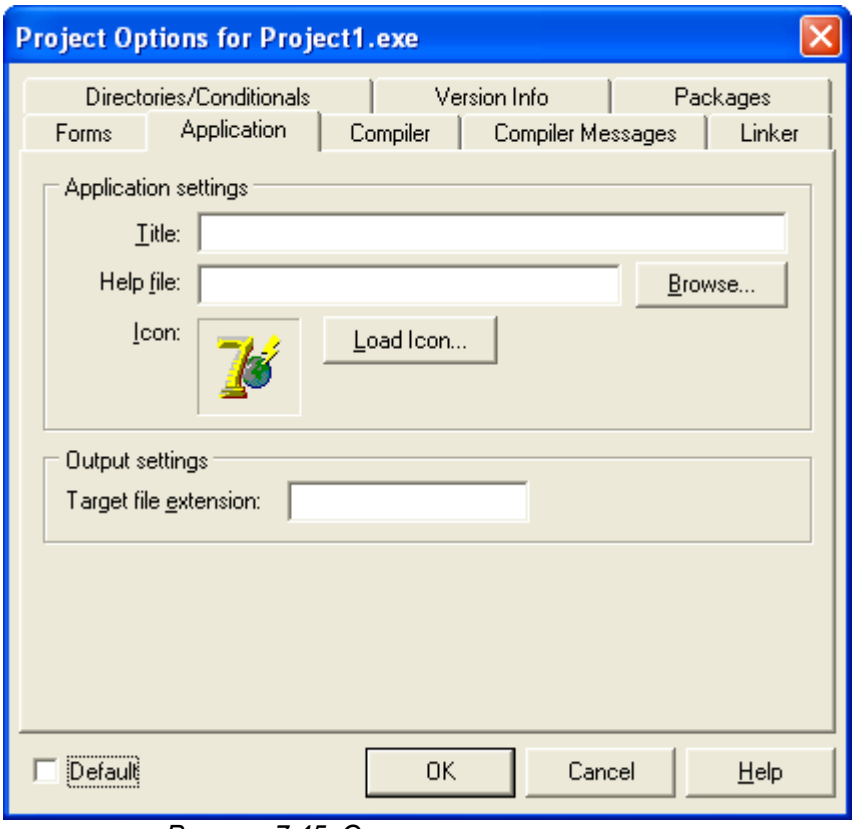


*Рисунок 7.45. Окно параметров проекта*

Объект **Application** имеет несколько полезных событий. Самые важные из них: **OnActivate**, **OnDeactivate**, **OnException**.

- **OnActionExecute** — происходит при выполнении любой команды в компоненте **ActionList** (см. главу 10).
- **OnActionUpdate** — происходит во время простоя программы для обновления состояния команд в компоненте **ActionList** (см. главу 10).
- **OnActivate** — происходит, когда программа получает фокус ввода, т.е. когда пользователь переключается на нее с другой программы.
- **OnDeactivate** — происходит, когда программа теряет фокус ввода, т.е. когда пользователь переключается с нее на другую программу.

- **OnException** — происходит, когда в программе возникает необработанная исключительная ситуация. Стандартный обработчик этого события вызывает метод **ShowException** для отображения окна сообщений с пояснением причины ошибки. Вы можете изменить реакцию на событие OnException, переписав его обработчик.
- **OnHelp** — происходит, когда пользователь вызывает справку.
- **OnHint** — происходит, когда курсор мыши наводится на компонент, содержащий всплывающую подсказку.
- **OnIdle** — периодически происходит во время простоя программы.
- **OnMessage** — происходит при получении программой сообщения операционной системы Windows.
- **OnMinimize** — происходит, когда пользователь сворачивает программу.
- **OnModalBegin** — происходит при отображении монопольной формы на экране.
- **OnModalEnd** — происходит при закрытии монопольной формы.
- **OnRestore** — происходит, когда пользователь восстанавливает свернутую программу.
- **OnSettingChange** — происходит при изменении настроек операционной системы, например, настроек экрана или региональных настроек.
- **OnShortCut** — происходит при нажатии клавиш на клавиатуре (еще до того, как в форме происходит событие **OnKeyDown**).
- **OnShowHint** — происходит непосредственно перед появлением любой всплывающей подсказки.

Из всех методов объекта **Application** мы упомянем лишь один — **Terminate**. Он выполняет штатное завершение приложения. Запомните, метод **Terminate** не вызывает немедленного завершения приложения, давая возможность всем формам корректно себя закрыть. Во время закрытия форм свойство **Terminated** имеет значение True.

При необходимости на помощь объекту **Application** спешат менее значительные "персоны": объекты **Screen**, **Printer** и **Clipboard**. Они также являются глобальными и создаются автоматически при запуске приложения (если, конечно, подключены стандартные модули, где они расположены).

## 7.7.2. Screen — объект, управляющий экраном

Каждая программа что-то выводит на экран, иначе она просто бесполезна. В среде Delphi экран трактуется как глобальный объект **Screen** класса TScreen, имеющий набор свойств. Многие из них жестко связаны с физическими характеристиками экрана (с "железом"), поэтому в большинстве случаев не доступны для записи. Обозначим самые важные свойства:

- **Width** и **Height** — ширина и высота экрана в пикселях.
- **ActiveForm** — активная форма (та, которая в текущий момент находится в фокусе ввода).
- **ActiveControl** — указывает компонент, который обладает фокусом ввода в активной форме.
- **Cursor** — управляет внешним видом указателя мыши для всех форм приложения.
- **Cursors** — список доступных указателей мыши.
- **DataModuleCount** — количество модулей данных, созданных приложением. Модуль данных — это нечто вроде невидимой формы, в которой можно размещать исключительно невизуальные компоненты. Перемещение невизуальных компонентов из формы в модуль данных может в ряде случаев улучшить структуризацию программы за счет отделения предметной программной логики от программной логики пользовательского интерфейса.
- **DataModules** — список всех модулей данных, созданных приложением.
- **DesktopWidth** и **DesktopHeight** — ширина и высота виртуального экрана (используется, когда к компьютеру подключено несколько мониторов).
- **DesktopLeft** и **DesktopTop** — позиция виртуального экрана на экране монитора.
- **DesktopRect** — координаты виртуального экрана.
- **Fonts** — список всех поддерживаемых шрифтов.
- **FormCount** — количество форм, созданных приложением.
- **Forms** — список всех форм, созданных приложением.
- **HintFont** — шрифт всплывающих подсказок.
- **IconFont** — шрифт подписей к значкам.
- **MenuFont** — шрифт текста в меню.
- **MonitorCount** — количество мониторов, подключенных к компьютеру.
- **Monitors** — список всех мониторов, подключенных к компьютеру.
- **PixelsPerInch** — количество пикселей в одном дюйме экрана монитора.

- **WorkAreaWidth** и **WorkAreaHeight** — ширина и высота рабочей области экрана (не включает панель задач). Если к компьютеру подключено несколько мониторов, то рассчитывается ширина и высота рабочей области на основном мониторе.
- **WorkAreaLeft** и **WorkAreaTop** — позиция рабочей области на экране монитора.
- **WorkAreaRect** — размеры и позиция рабочей области на экране монитора.

В качестве примера использования объекта **Screen** приведем фрагмент, устанавливающий указателю мыши вид песочных часов на время выполнения какой-либо длительной операции:

```
Screen.Cursor := crHourGlass;
try
  { Длительная операция }
finally
  Screen.Cursor := crDefault;
end;
```

### 7.7.3. Mouse — объект, представляющий мышь

Вряд ли сейчас можно встретить компьютеры без миниатюрного "хвостатого" устройства, называемого мышью. Для работы с ним в среде Delphi есть специальный объект **Mouse**: **TMouse**, автоматически добавляемый в программу при подключении модуля **Controls**. Перечислим наиболее важные свойства этого объекта:

- **Capture** — содержит описатель окна, захватившего мышь для монопольного использования (это объект операционной системы Windows).
- **CursorPos** — позиция указателя мыши.
- **DragImmediate** — определяет, когда начинается буксировка: значение True — немедленно, значение False — после того, как указатель мыши переместиться на **DragThreshold** позиций при удерживаемой кнопке мыши.
- **DragThreshold** — количество пикселей, на которые необходимо переместить указатель при нажатой кнопке мыши, чтобы началась буксировка.
- **IsDragging** — проверяет, идет ли в данный момент процесс буксировки.
- **MousePresent** — проверяет, подключена ли мышь к компьютеру.
- **WheelPresent** — проверяет, есть ли у мыши колесико.
- **WheelScrollLines** — количество логических строк, на которые смещается страница при прокрутке колесика мыши на один шаг.

### 7.7.4. Printer — объект, управляющий принтером

Большинство программ выводят некоторый текст или рисунки на печатающее устройство. Для этого полезного дела в среде Delphi имеется специальный объект **Printer**. Он становится доступен после подключения модуля Printers. Если вы включите этот модуль в проект, сразу после старта будет порожден объект **Printer** класса TPrinter. Его свойства и методы дают вам весьма неплохие возможности для печати из приложения на все виды принтеров. Однако, тема эта заслуживает отдельной главы (см. гл. 10).

### 7.7.5. Clipboard — объект, управляющий Буфером обмена

Каждый, кто работал с текстом, знает, какая это великолепная штука — *Буфер обмена* (Clipboard). Напомним, что это буфер, куда можно что-то положить (например, текст или рисунок), а потом взять это оттуда. За операции с Буфером обмена в среде Delphi отвечает глобальный объект **Clipboard** класса TClipboard. Он расположен в модуле Clipbrd. О том, как объект **Clipboard** используется практически, подробно рассказано в гл. 8.

## 7.8. Итоги

Время потрачено не зря! Вы узнали о проекте все:

- что он собой представляет и из каких частей состоит (файлы описания форм, файлы программных модулей, главный файл проекта и др.);

- как открывать, сохранять, выполнять проект и управлять им с помощью окна **Project Manager**;
- что есть форма, как изменять ее стиль, размер, местоположение, цвет, как переключаться с главной формы на второстепенную и наоборот и т.д.
- что есть компонент, откуда его взять, куда поместить, как навести порядок в группе компонентов;
- кто управляет приложением изнутри (объект Application) и кто ему в этом помогает (объекты Screen, Printer, Clipboard).

Да, трудновато все это было усвоить, но надо. Тяжело в учении — легко в бою. Утешив себя этой истиной, перейдем к изучению важнейших элементов пользовательского интерфейса — меню, панели инструментов и строки состояния.