А. И. Новиков

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ Delphi и Pascal

Часть 2

Учебно-методическое пособие

Санкт-Петербург
2025

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Санкт-Петербургский государственный университет промышленных технологий и дизайна»
Высшая школа технологии и энергетики

А.И. Новиков

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ Delphi и Pascal

Часть 2

Учебно-методическое пособие

Утверждено Редакционно-издательским советом ВШТЭ СПбГУПТД

Санкт-Петербург
2025

УДК 004.432
ББК 32.973
Н 731

Рецензенты:
кандидат технических наук, доцент, заведующий кафедрой АТПиП Высшей школы технологии и энергетики
Санкт-Петербургского государственного университета промышленных технологий и дизайна
Д. А. Ковалёв;
кандидат технических наук, доцент, председатель учебно-методической комиссии факультета ИТиУ Санкт-Петербургского государственного технологического института (технического университета)
В. В. Куркина

Новиков, А. И.
Н 731 Алгоритмизация и программирование. Delphi и Pascal. Часть 2: учебно-методическое пособие / А. И. Новиков. — СПб.: ВШТЭ СПбГУПТД, 2025. — 108 с.

Учебно-методическое пособие соответствует программам и учебным планам дисциплины «Алгоритмизация и программирование» для студентов, обучающихся по направлению подготовки 09.03.03 «Прикладная информатика». В пособии изложены принципы работы в среде Delphi, основы языка Pascal и распространенные парадигмы программирования. Приведены примеры разработки программ. Учебно-методическое псобие содержит разделы для самостоятельного углубленного изучения.

Пособие предназначено для подготовки бакалавров очной и заочной форм обучения. Отдельные разделы пособия могут быт полезны магистрантам и аспирантам.

УДК 004.432
ББК 32.973

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Событийно-ориентированное программирование – парадигма программирования, в которой выполнение программы определяется событиями (Events) – т.е. действиями пользователя (клавиатура, мышь, сенсорный экран), а также сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета). Событийно-ориентированное программирование применяется при построении пользовательских интерфейсов, или, например, при создании игр, в которых осуществляется управление множеством объектов.

Графический интерфейс пользователя (GUI) – система средств для взаимодействия пользователя с электронными устройствами, основанная на представлении всех доступных пользователю системных объектов в виде графических компонентов экрана (окон, значков, меню, кнопок, списков и т. п.). GUI является стандартной составляющей большинства современных операционных систем, таких как: Microsoft Windows, Mac OS, Linux, Android, iOS и др.

Объектно-ориентированное программирование (ООП) – парадигма программирования, основанная на концепции объектов. Итоговые программы создаются из объектов, взаимодействующих друг с другом. Компоненты (такие как кнопки, поля ввода, чекбоксы, радиокнопки и др.) являются одним из вариантов Объектов.

* Пособие содержит разделы для самостоятельного изучения, их названия отмечены звездочкой.

3. СОБЫТИЙНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Событийно-ориентированное программирование (event-driven programming), или иногда кратко, **Событийное программирование** – парадигма программирования, в которой выполнение программы определяется событиями (**Events**) – т.е. действиями пользователя (клавиатура, мышь, сенсорный экран), а также сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета). Событийно-ориентированное программирование применяется при построении пользовательских интерфейсов, или, например, при создании игр, в которых осуществляется управление множеством объектов.

Несмотря на то, что **Delphi** поддерживает различные парадигмы, включая Модульное программирование *(с которым мы уже встречались)* или Объектно-ориентированное программирование *(с которым мы еще познакомимся дальше)*, но не они являются основой **Delphi**, значительно упрощающей жизнь программисту. Именно Событийно-ориентированное программирование позволяет реализовать концепцию «быстрой разработки приложений» (**RAD**).

На самом деле, мы уже сталкивались с событиями *(хотя сами этого и не замечали)*, это было событие **Click** («Клик», «Щелчок») для кнопки. Когда при создании «Калькулятора» в редакторе Delphi мы дважды кликали мышкой по кнопке, то у нас автоматически создавалась новая процедура, имеющая вид:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //Здесь писался код для кнопки
end;
```

После запуска программы (**F9**), если мы осуществляли клик мышкой по этой кнопке, то выполнялся соответствующий код.

Внутри подобных процедур пишется код, который выполнялся «если» была нажата кнопка, но есть одно «но», у нас в коде нигде не написано команды «**if**», которая переводится как «**если**»… Поэтому мы не должны говорить, «Если нажата кнопка».

Представим на секунду, что у нас не существует никаких событий, есть только кнопка Button1, программа, повторяемая в бесконечном цикле, и три функции:

X: Integer;
Y: Integer;
LeftBth: Boolean;

Первые две позволяют получить текущие координаты мышки, а LeftBtn сообщает нам, нажата кнопка мышки в настоящий момент или нет.

Тогда, прежде чем написать код для кнопки, нам вначале **самостоятельно** нужно проверить, нажата ли кнопка мышки, а если нажата, то нужно определить еще что она нажата именно внутри этой кнопки Button1. Такой код будет иметь следующий вид:

```
if LeftBth then
begin
  if (X >= Button1.Left) and (X < Button1.Left + Button1.Width) and
     (Y >= Button1.Top) and (Y < Button1.Top + Button1.Height) then
  begin
    //Код для кнопки
  end;

  //...
end;
```

Так много лишнего нам пришлось написать всего для одной кнопки. Причем в конце мы не зря оставили многоточие: туда нужно будет писать код для других кнопок… Сколько их было для «Калькулятора»?

Кроме того, на самом деле это даже неполный код. В приведенном примере при удерживании кнопки мышки команда будет выполняться многократно в цикле, но нам нужно чтобы на каждое нажатие мышки, команда выполнилась только один раз (причем неважно сколько мы удерживаем кнопку). *Мы не будем даже пытаться доработать код до этого усложненного варианта.*

К счастью, в **Delphi** все это писать и не нужно, а создаваемые средой процедуры правильно называть **не** «Если нажата кнопка», а «**Когда** нажата кнопка». А еще точнее «по» или «после», например, «По щелчку мышкой» или «После движения мышки».

В действительности в **Delphi** осуществляются все те проверки координат, которые мы рассмотрели, но нам их писать не нужно, т.к. они уже реализованы разработчиками **Delphi** и ОС **Windows**, и названы событием **OnClick**, которым мы и пользовались ранее. В этом и заключается смысл Событийно-ориентированного программирования.

3.1. События в Инспекторе объектов

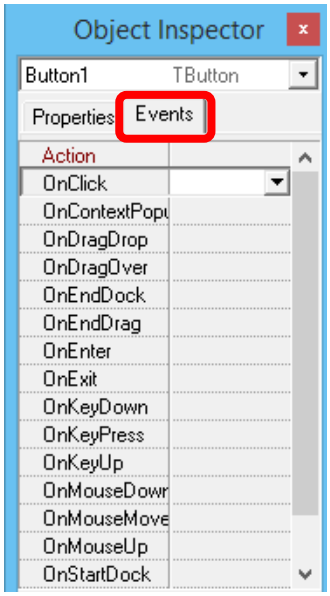


Рисунок 3.1 – Пример событий для кнопки

События для компонентов можно найти в Инспекторе объектов, но нужно перейти на вкладку «**Events**» (рис. 3.1). Имена событий в **Delphi** принято

начинать с приставки «**On**», которая и переводится как «по» или «после». Например, **OnClick** – «По щелчку», или **OnMouseMove** – «После движения мышки».

Для использования требуемого события необходимо дважды кликнуть по нему мышкой в Инспекторе объектов. При этом для данного события будет создан требуемый код (называемый «**Обработчик события**»), а курсор ввода будет помещен между соответствующими **begin** и **end**, что позволяет сразу начать вводить команды, выполняемые при данном событии. *То же самое мы уже видели, когда дважды кликали по* ***Button*** *при создании «Калькулятора».* После этого в Инспекторе объектов для данного события отобразится имя его обработчика. При желании можно его переименовать *(но обычно это делать не рекомендуется)*. Кроме того, в Инспекторе объектов можно выбирать Обработчики событий из выпадающего списка (если они ранее уже были созданы), это позволяет привязать один обработчик событий сразу к нескольким компонентам.

3.2. События визуальных компонентов

Основные события визуальных компонентов представлены в таблице 1.

Таблица 1 – Основные события визуальных компонентов

Событие	Описание
OnClick: TNotifyEvent;	Клик («щелчок») мышкой по компоненту. У кнопки Button это событие также срабатывает, если нажать «Enter» или «Пробел» в тот момент, когда фокус ввода находится на кнопке *(обычно при этом кнопка выделена пунктирной рамкой)*. Только один компонент на форме может иметь фокус.
OnDblClick: TNotifyEvent;	Двойной клик мышкой. *Имеется не у всех компонентов.*
OnContextPopup: TContextPopupEvent;	Всплывающее меню. Фактически это нажатие правой клавиши мышки. В отличие от OnClick и OnDblClick, возвращает еще и координаты нажатия мышки. *Координаты для данного события доступны по именам MousePos.X и MousePos.Y.*
OnMouseMove: TMouseMoveEvent;	Движение курсора мышки над компонентом (при этом нажатие клавиш мышки не обязательно, хотя и возможно). Возвращает координаты курсора мышки (X, Y). Также возвращает информацию о том, нажата ли какая-то клавиша мышки, и удерживаются ли при этом клавиши Alt, Ctrl или Shift.

Продолжение табл. 1

Событие	Описание
OnMouseDown: TMouseEvent; **OnMouseUp**: TMouseEvent;	Нажатие или отпускание клавиши мышки (не обязательно левой). Это более подробный случай для OnClick. Позволяет разделить клик мышкой на нажатие и отпускание клавиши. Возвращает координаты курсора (X, Y), а также информацию о том, какая клавиша мышки нажата и удерживаются ли при этом клавиши Alt, Ctrl или Shift.
OnKeyPress: TKeyPressEvent;	Нажатие клавиши на клавиатуре. Возвращает символ нажатой клавиши. *Фокус ввода должен находиться на компоненте (обычно это отображается как мигающий курсор ввода текста, или как пунктирная рамка вокруг кнопки)*.
OnKeyDown: TKeyEvent; **OnKeyUp**: TKeyEvent;	Нажатие или отпускание клавиши на клавиатуре. Это более подробный случай для OnKeyPress. Позволяет разделить нажатие клавиши клавиатуры на нажатие и отпускание. Возвращает код нажатой клавиши, а также информацию о том, удерживаются ли при этом клавиши Alt, Ctrl или Shift
OnEnter: TNotifyEvent; **OnExit**: TNotifyEvent;	Срабатывает, когда данный компонент получает фокус или теряет его. *Фокус ввода отображается как мигающий курсор ввода текста или как пунктирная рамка вокруг выбранной кнопки.* Для переключения фокуса компонентов, помимо мышки, также можно использовать кнопку «Tab» *(но кнопка «Tab» не будет работать для компонента, для которого установлено значение TabStop := False).* Только один компонент на форме может иметь фокус.
OnChange: TNotifyEvent;	«Изменение». Срабатывает при изменении содержимого компонента. Например, при изменении текста в Edit, Memo или RichEdit, или изменение численного значения в SpinEdit.
OnSelectionChange: TNotifyEvent;	Срабатывает при изменении положения каретки (курсора для ввода текста). *Есть только у RichEdit.*

Окончание табл. 1

Событие	Описание
OnMouseWheel: TMouseWheelEvent; **OnMouseWheelDown**: TMouseWheelUpDownEvent; **OnMouseWheelUp**: TMouseWheelUpDownEvent;	Вращение колесика мышки. Есть далеко не у всех компонентов. Имеется, например, у RichEdit. *При этом компонент должен быть в фокусе, т.е. курсор ввода текста должен быть на нем.*
OnMouseEnter: TNotifyEvent; **OnMouseLeave**: TNotifyEvent;	Срабатывают, когда курсор мышки заходит в область над компонентом или выходит из нее. *В старых версиях (таких как Delphi 7) они скрыты почти для всех компонентов, но в новых версиях (таких как Delphi XE8) доступны для большинства компонентов*

Уведомления

Самым простым типом для событий являются «Уведомления» (**Notify** – «Уведомить», «Поставить в известность»):

TNotifyEvent = procedure(Sender: TObject) of object;

Такое событие не передает никаких дополнительных данных, кроме адреса отправителя (**Sender**). Для него Delphi генерирует следующий обработчик:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

!!! Стоит обратить внимание, что в создаваемом **автоматически** имени такой процедуры, всегда будет указано имя события (в данном случае «**Click**»), а перед ним имя компонента, для которого наступает это событие (в данном случае это «**Button1**»).

Движение мышки

TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState;
X, Y: Integer) of object;

Данный вид события не только «уведомляет» о том, что что-то произошло, но и сообщает, где это произошло, передавая координаты курсора.

Например, отобразим в заголовке окна, координаты курсора мышки, который двигается над формой TForm1. Для этого в обработчике события **OnMouseMove** этой формы напишем:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  Caption := IntToStr(X) + ':' + IntToStr(Y);
end;
```

Координаты отсчитываются от верхнего левого угла окна (рис. 3.2).

!!! Этот пример с **OnMouseMove** стоит запомнить, т.к. он (как и следующие примеры с мышкой) уже скоро понадобится нам при разработке «Графического редактора» (раздел **5.6**).

Рисунок 3.2 – Координаты курсора в заголовке

Также данное событие сообщает нам параметр **Shift**, типа:

TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble);

где ssShift, ssAlt, ssCtrl – сообщают, удерживаются ли в данный момент клавиши Shift, Alt или Ctrl;

ssLeft, ssRight, ssMiddle – сообщают, удерживаются ли в данный момент какие-то из клавиш мышки (левая, правая или средняя);

ssDouble – сообщает, что был двойной клик мышкой.

Стоит обратить внимание, что **TShiftState** объявлен как **SET** («Набор»). К нему мы обращаемся **не** через «равенство» или «точку», здесь необходимо использовать оператор **in** («в»), т.е. мы проверяем «входит ли значение **в** набор?».

Например, опять будем отображать координаты курсора при движении мышки, но теперь будем обновлять текст в заголовке только когда удерживается клавиша **Alt**:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if ssAlt in Shift then
  begin
    Caption := IntToStr(X) + ':' + IntToStr(Y);
  end;
end;
```

Нажатие кнопок мышки

TMouseEvent = procedure(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer) of object;

Например, отобразим координаты нажатия кнопки мышки на форме:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Caption := IntToStr(X) + ':' + IntToStr(Y);
end;
```

Аналогично отобразим координаты отпускания мышки, но будем добавлять их к уже имеющимся в заголовке начальным координатам (рис. 3.3):

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Caption := Caption + ' -> ' + IntToStr(X) + ':' + IntToStr(Y);
end;
```

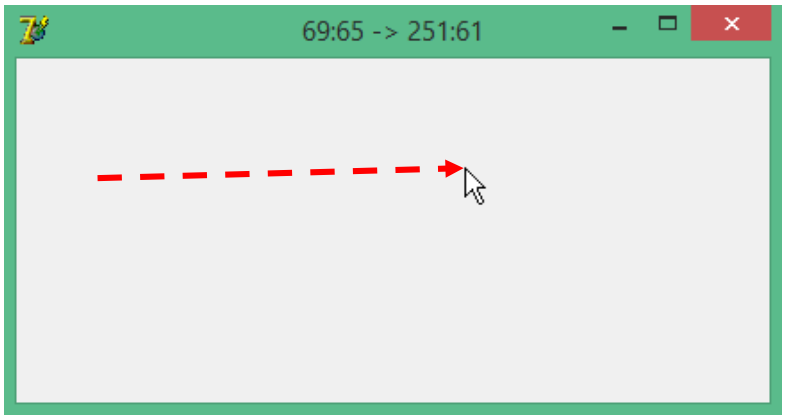


Рисунок 3.3 – Координаты отпускания курсора

Хорошая новость: на самом деле, нет необходимости выписывать все типы событий или заучивать их. Т.к. после двойного клика по требуемому событию, **Delphi** автоматически создаст для нас обработчик для этого события – процедуру, в заголовке которой будут видны все передаваемые параметры. Нам нужно лишь помнить, какие события являются только уведомлениями («**Notify**»), а какие сообщают нам дополнительную информацию (**не** «**Notify**»).

Ограничение ввода символов с клавиатуры и замена символов

Обработчик событий **OnKeyPress** возвращает символ нажатой клавиши **Key**. Но при желании его можно изменить (до того, как он будет отображен). Например, можно заменять точки на запятые при вводе в **Edit**. Кроме того, можно вообще запретить некоторые символы, выводя вместо них:

Key := #0;

Рассмотрим следующий пример, позволяющий вводить в поле **Edit** только числа:

```
{----------------------- Разрешить вводить только числа -----------------------}
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  //Замена точки на запятую
  if Key = '.' then Key := ',';
  //Разрешить ввод только указанных символов
  if not (Key in ['0'..'9', ',', '-', '+', 'e', 'E', #8]) then Key := #0;
end;
```

Здесь **Key** сравнивается со списком значений. Запись вида:

Key **in** ['a', 'b', 'c']

равносильна следующей записи:

(Key = 'a') **or** (Key = 'b') **or** (Key = 'c')

!!! Кроме того, можно задавать диапазоны значений через «многоточие» (две точки**!**).

Стоит обратить внимание на то, что для ввода чисел недостаточно только цифр и запятой, нам необходим, как минимум, еще «минус». Кроме того, числа можно вводить через 'E' (или 'e').

Символ с номером **#8** означает клавишу «**Backspace**». Если не включить его в список разрешенных, то мы потеряем возможность стирать последний символ. *Но это не распространяется на клавиши «**Delete**» и «←», «→», «↑», «↓». Они будут работать всегда.*

Пример для OnChange

Событие **OnChange** наступает при изменении текста в поле **Edit**. Это может быть использовано для расчета значений, зависящих от этого поля, «на лету» (т.е. без необходимости нажимать кнопу расчета). Например, в Edit1 и Edit2 вводятся два числа, а в Edit3 выводится их сумма (рис. 3.4.):

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Edit3.Text := IntToStr(StrToInt(Edit1.Text) + StrToInt(Edit2.Text));
end;
```

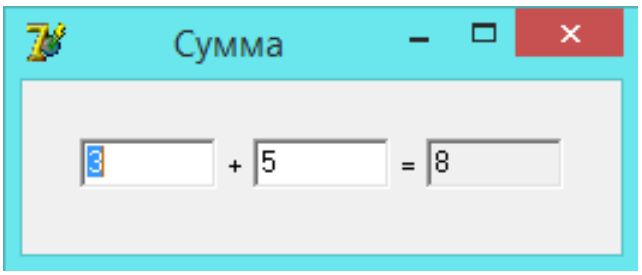


Рисунок 3.4 – Вывод результата «на лету»

Для Edit2 необходимо выполнять тотже код. Но его не нужно писать еще раз**!** Это тот самый случай, когда можно выбрать уже существующий обработчик из выпадающего списка (в Инспекторе объектов).

!!! Стоит обратить внимание, что в поля **Edit** можно ввести текст, который **не** является числом, что приведет к ошибке**!** Например, если ввести букву, то сразу выскочит сообщение об ошибке. Аналогично, ошибка произойдет и, если, полностью стереть текст (Edit1.Text :="). Т.е. для правильной работы данной программы, ее необходимо дополнить условиями с проверками. Также можно ограничить вводимые в **Edit** символы, при помощи **OnKeyPress**.

Кроме того, можно использовать событие **OnChange** для изменения цвета текста, в зависимости от введенного в него значения, например:

```
procedure TForm1.Edit1Change(Sender: TObject);
var f: Double;
begin
  //Выделяем красным, если не является числом
  if TryStrToFloat(Edit1.Text, f) then Edit1.Font.Color := clBlack
                                  else Edit1.Font.Color := clRed;
end;
```

Пример для OnSelectionChange

Будем выводить координаты каретки для ввода текста (номер строки и номер символа в строке) в заголовок окна (рис. 3.5).

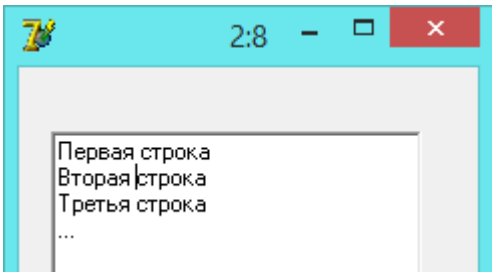


Рисунок 3.5 – Координаты каретки в заголовке

Это необходимо делать при каждом изменении положения каретки, т.е. при наступлении **OnSelectionChange**, в обработчике которого напишем:

```
procedure TForm1.RichEdit1SelectionChange(Sender: TObject);
begin
  Caption := IntToStr(RichEdit1.CaretPos.Y+1) + ':' +
             IntToStr(RichEdit1.CaretPos.X+1);
end;
```

!!! Также можно обновлять данные о положении каретки не при каждом изменении ее положения, а по таймеру, например, каждые 0,2 сек.

Конечно, это был всего лишь пример, и в реальности никто не будет выводить координаты в заголовок окна. Теперь будем выводить информацию о положении каретки в Строке состояния **StatusBar** (рис. 3.6).

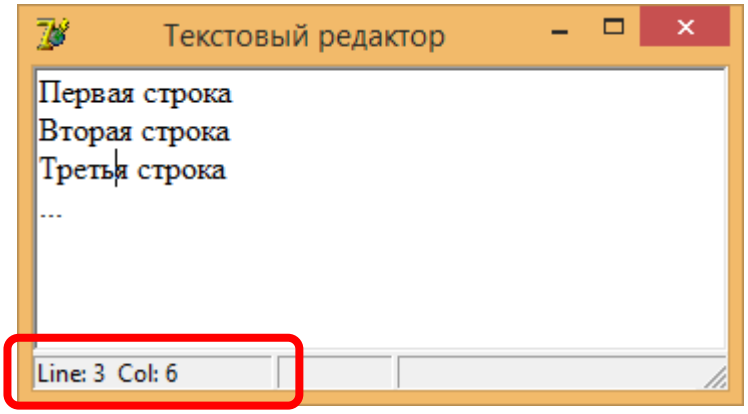


Рисунок 3.6 – Координаты каретки в панели состояния

Для этого необходимо:

– добавить на форму компонент **TStatusBar** (расположен на вкладке «**Win32**»), который автоматически выравнивается внизу окна;
– дважды кликнуть по этой панели;
– в появившемся окне (рис. 3.7) добавить три панели, из которых будет состоять «Строка состояния» (см. рис. 3.8);

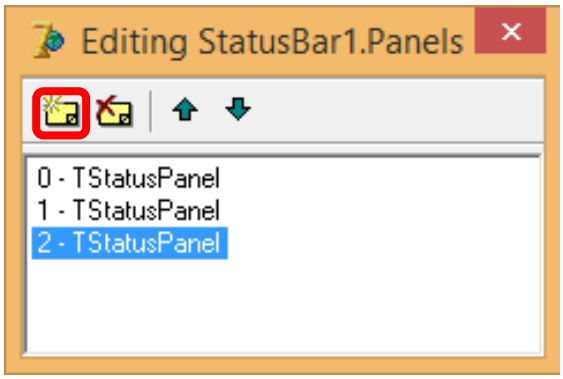


Рисунок 3.7 – Добавление панелей

– для каждой из этих панелей (кроме последней) необходимо настроить **ширину** (рис. 3.8). Также можно написать для них начальный текст.

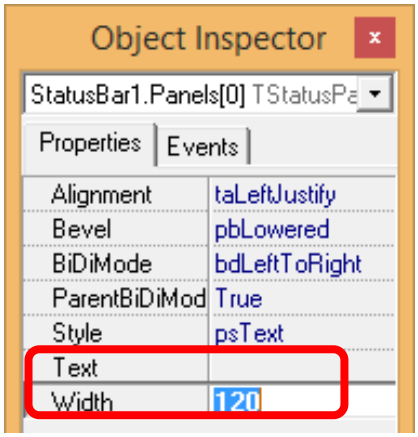


Рисунок 3.8 – Настройка панелей

Тогда в итоге получим следующий код:

```
{---------------------------- Положение каретки ------------------------------}
procedure TForm1.RichEdit1SelectionChange(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := 'Line: ' + IntToStr(RichEdit1.CaretPos.Y+1)
                             + '  Col: ' + IntToStr(RichEdit1.CaretPos.X+1);
end;
```

!!! Этот пример с **OnSelectionChange** и **StatusBar** стоит запомнить, т.к. он далее понадобится нам при доработке «Текстового редактора» (раздел **3.6**).

3.3. События невизуальных компонентов

События для Action

Действия (Action) имеют собственные события, и не поддерживают события, перечисленные выше. События **TAction** представлены в таблице 2.

Таблица 2 – События TAction

Событие	Описание
OnExecute: TNotifyEvent;	Выполнение текущего действия. Для всех **собственных** действий его применение **обязательно**, иначе они будут недоступны (будут серого цвета). Когда действие связывается с кнопкой или пунктом меню, то **OnExecute** выполняется вместо их **OnClick**.
OnUpdate: TNotifyEvent;	Позволяет обновлять состояние текущего действия, например, его доступность (**Enabled**), видимость (**Visible**), или значение **Checked**. Не является обязательным.
OnHint: THintEvent;	Выполняется перед отображением всплывающей подсказки. Позволяет изменить текст подсказки или запретить ее показ. Применяется редко.

Например, будем отображать сообщение «**О программе**» (рис. 3.9) при выборе соответствующего действия в меню:

```
{------------------------------ О программе --------------------------------}
procedure TForm1.About1Execute(Sender: TObject);
begin
  ShowMessage('Текстовый редактор Note' + #13#10 + '© Alexandr Novikov, 2023');
end;
```

!!! Этот пример с **OnExecute** и **ShowMessage** стоит запомнить, т.к. он далее понадобится нам при доработке «Текстового редактора» (раздел **3.6**).

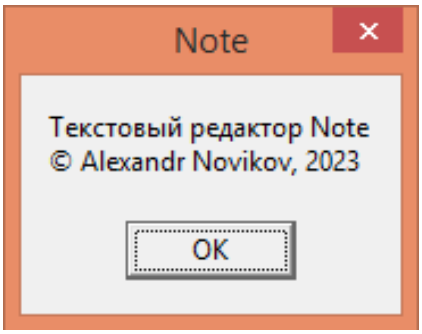


Рисунок 3.9 – Окно «О Программе»

Другим примером будет Действие (Кнопка) для выделения текста в **RichEdit** красным шрифтом. Если ранее выделенный текст уже был красным, то его необходимо сделать обратно черным:

```
{------------------------------ Красный текст ------------------------------}
procedure TForm1.RedFontActionExecute(Sender: TObject);
begin
  if RichEdit1.SelAttributes.Color <> clRed
    then RichEdit1.SelAttributes.Color := clRed
    else RichEdit1.SelAttributes.Color := clBlack;
end;
```

При этом мы хотим, чтобы кнопка имела возможность отображаться зажатой, когда выделенный текст уже имеет красный цвет *(по аналогии с кнопкой «**Жирный**» в **Word**)*:

```
{-------------------------------------------------------------------------------}
procedure TForm1.RedFontActionUpdate(Sender: TObject);
begin
  RedFontAction.Checked := (RichEdit1.SelAttributes.Color = clRed);
end;
```

События для Action с диалоговыми окнами

События для действий, имеющих встроенные диалоговые окна, значительно отличаются (рис. 3.10) от обычных **TAction**. События **Action** со встроенными диалогами представлены в таблице 3.

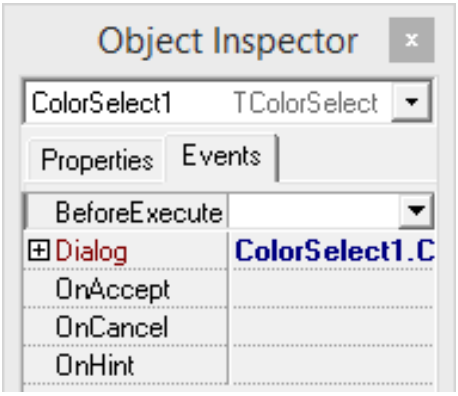


Рисунок 3.10 – События для действия TColorSelect

Таблица 3 – События Action со встроенными диалогами

Событие	Описание
OnAccept: TNotifyEvent;	Выполняется, когда в диалоговом окне нажата кнопка «Ok». *Для почти всех действий с диалогами его применение обязательно, иначе они не будут выполнять ничего полезного, кроме открытия диалога.*
BeforeExecute: TNotifyEvent; *{в нарушение правил, пишется без приставки 'On'}*	Выполняется непосредственно перед открытием диалогового окна. С его помощью можно задать начальные значения для диалога.
OnCancel: TNotifyEvent;	Выполняется, когда в диалоговом окне нажата кнопка «Отмена». Используется редко.

Например, для действия выбора цвета (**TColorSelect**) напишем следующий код:

```
{--------------------------- Выбрать цвет текста ----------------------------}
procedure TForm1.ColorSelect1Accept(Sender: TObject);
begin
  RichEdit1.SelAttributes.Color := ColorSelect1.Dialog.Color;
end;
```

Теперь после нажатия в диалоговом окне кнопки «**OK**», выбранный цвет применяется к тексту, который выделен в **RichEdit**. То есть цвет переносится из Диалога в **RichEdit**.

Но в этом случае при открытии Диалога, он **не** будет отображать цвет выделенного в настоящий момент текста (при первом открытии всегда будет черным, а в следующие разы будет отображаться цвет, выбранный в прошлый раз). Т.е. теперь наоборот, требуется переносить цвет из **RichEdit** в Диалог. Тогда необходимо создать также обработчик для события **BeforeExecute**:

```
{------------------------------------------------------------------------------}
procedure TForm1.ColorSelect1BeforeExecute(Sender: TObject);
begin
  ColorSelect1.Dialog.Color := RichEdit1.SelAttributes.Color;
end;
```

Аналогично для действия выбора шрифта (**TFontEdit**) напишем:

```
{------------------------------ Выбрать шрифт -------------------------------}
procedure TForm1.FontEdit1Accept(Sender: TObject);
begin
  //Цвет и размер шрифта
  RichEdit1.SelAttributes.Color   := FontEdit1.Dialog.Font.Color;
  RichEdit1.SelAttributes.Size    := FontEdit1.Dialog.Font.Size;
  //Жирный, курсив, зачеркнутый, подчеркнутый
  RichEdit1.SelAttributes.Style   := FontEdit1.Dialog.Font.Style;
  //Имя шрифта и набор символов
  RichEdit1.SelAttributes.Name    := FontEdit1.Dialog.Font.Name;
  RichEdit1.SelAttributes.Charset := FontEdit1.Dialog.Font.Charset;
end;
{------------------------------------------------------------------------------}
procedure TForm1.FontEdit1BeforeExecute(Sender: TObject);
begin
  //Цвет и размер шрифта
  FontEdit1.Dialog.Font.Color   := RichEdit1.SelAttributes.Color;
  FontEdit1.Dialog.Font.Size    := RichEdit1.SelAttributes.Size;
  //Жирный, курсив, зачеркнутый, подчеркнутый
  FontEdit1.Dialog.Font.Style   := RichEdit1.SelAttributes.Style;
  //Имя шрифта и набор символов
  FontEdit1.Dialog.Font.Name    := RichEdit1.SelAttributes.Name;
  FontEdit1.Dialog.Font.Charset := RichEdit1.SelAttributes.Charset;
end;
```

Диалоги для работы с файлами

Аналогично для действий сохранения и загрузки файла необходимо создать обработчик для их события **OnAccept**. Например, если нам требуется сохранять/загружать содержимое **RichEdit**, то пишется следующий код:

```
//Сохранить в файл
RichEdit1.Lines.SaveToFile(FileSaveAs1.Dialog.FileName);

//Загрузить из файла
RichEdit1.Lines.LoadFromFile(FileOpen1.Dialog.FileName);
```

Если используется **Memo**, то делается точно так же (только **RichEdit** заменяется на **Memo**).

Событие таймера

Невизуальный компонент **TTimer** расположен на вкладке «**System**». Таймер имеет единственное событие, представленное в таблице 4.

Таблица 4 – События таймера

Событие	Описание
OnTimer: TNotifyEvent;	Выполняется через равные промежутки времени. Промежутки задаются свойством **Interval** через Инспектор объектов. Время измеряется в **милли**секундах. По умолчанию установлено значение **1000** (1 секунда). Таймер можно отключать, устанавливая его свойство **Enabled** := False. *Данный таймер **не** предназначен для точных замеров времени. Значения интервала меньше 100 задавать нет смысла.*

Например, будем сдвигать кнопку на **10** пикселей каждые **0,5** секунды. Тогда в обработчике события таймера необходимо написать код:

```
Button1.Left := Button1.Left + 10;
```

Также нужно настроить таймер, установив для него **Interval** := 500.

3.4. События формы

Форма содержит многие из перечисленных ранее событий визуальных компонентов, таких как OnClick, OnDblClick, OnContextPopup, OnMouseMove, OnMouseDown, OnMouseUp, OnMouseWheel, OnMouseWheelDown, OnMouseWheelUp, OnKeyPress, OnKeyDown, OnKeyUp. Но у формы имеются и собственные события (табл. 5), которых нет у других компонентов.

Таблица 5 – Основные события формы

Событие	Описание
OnCreate: TNotifyEvent;	Создание формы. Выполняется до того, как окно будет показано в первый раз. Здесь можно задать начальные значения переменным, загрузить исходные данные из файла, создать компоненты *(при их динамическом создании)* и т.п.
OnCanResize: TCanResizeEvent;	Запрос на изменение размеров окна. Наступает при каждой попытке изменить размеры окна. Здесь можно запретить изменение размера, установив **Resize** := False; или задать новые размеры окна **NewWidth** и **NewHeight**.

Окончание табл. 5

Событие	Описание
OnClose: TCloseEvent;	Уничтожение формы. Выполняется при закрытии программы *(для Главной формы, а для остальных может происходить и раньше)*. Здесь можно сохранить итоговые данные в файл, уничтожить компоненты *(которые были созданы динамически)* и т.п.
OnCloseQuery: TCloseQueryEvent;	Запрос на закрытие. Наступает при каждой попытке закрыть окно. Здесь, например, можно отобразить диалоговое окно подтверждения закрытия, или предложить сохранить файл перед закрытием. Действия в **OnCloseQuery** выполняются раньше, чем **OnClose**. Это последняя возможность не дать закрыть программу, для этого необходимо указать значение переменной **CanClose** := False (по умолчанию **CanClose** := True).

Например, разместим на форме единственный CheckBox (рис. 3.11) и напишем следующий код:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose:
Boolean);
begin
  if CheckBox1.Checked then CanClose := False;
end;
```

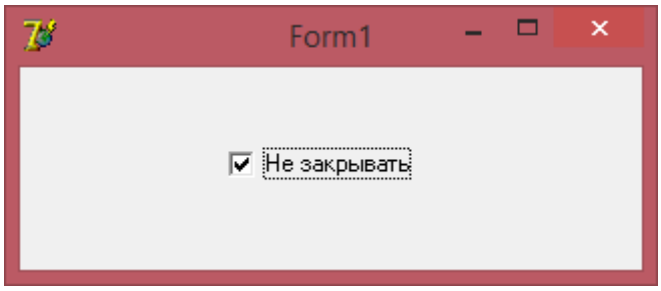


Рисунок 3.11 – Пример для OnCloseQuery

Данная программа не позволяет закрыть окно до тех пор, пока не будет снята галочка.

3.5. Диалоговые окна Application.MessageBox

На самом деле, Application.MessageBox не имеет прямого отношения к Событийно-ориентированному программированию. В эту Главу он попал, т.к. является хорошей демонстрацией для события формы OnCloseQuery.

Ранее мы уже рассмотрели процедуру **ShowMessage**, которая отображала диалоговое окно. Но у нее был всего один параметр – текст, отображаемый в

окне. Для нее нельзя было изменить название окна, и это окно содержало только единственную кнопку «**OK**».

Больше возможностей можно получить, используя функцию **Application.MessageBox**, объявленную как:

function **MessageBox**(Text, Caption: PChar; Flags: Longint): Integer;

где **Text** и **Caption** – текст окна и заголовок окна;

Flags – сумма флагов с различными дополнительными параметрами.

Флаги отвечают за количество кнопок (табл. 6) и изображение в окне (табл. 7).

Таблица 6 – Кнопки диалогового окна

Значение	Кнопки
MB_OK	*Единственная кнопка* «ОК»
MB_OKCANCEL	«ОК», «Отмена»
MB_YESNO	«Да», «Нет»
MB_YESNOCANCEL	«Да», «Нет», «Отмена»
MB_RETRYCANCEL	«Повтор», «Отмена»
MB_ABORTRETRYIGNORE	«Прервать», «Повтор», «Пропустить»

Таблица 7 – Изображения диалогового окна

Иконка	Значение	Описание
	**MB_ICONWARNING**	Восклицательный знак (замечание, предупреждение)
	**MB_ICONINFORMATION**	Буква «i» в круге (информация)
	**MB_ICONQUESTION**	Знак вопроса (вопрос, ожидание ответа)
	**MB_ICONERROR**	Крест на красном фоне (ошибка, стоп, запрет)

Изображение не является обязательным для диалогового окна. Его можно вообще не указывать.

Кроме того, при помощи флагов можно задать кнопку по умолчанию, которая будет выбрана при открытии окна диалога (табл. 8). Для использования варианта, выбранного по умолчанию, в окне достаточно будет нажать клавишу «**Enter**».

Таблица 8 – Кнопки по умолчанию

Значение	Кнопка
MB_DEFBUTTON1	Выбрана первая кнопка (значение по умолчанию)
MB_DEFBUTTON2	Выбрана вторая кнопка
MB_DEFBUTTON3	Выбрана третья кнопка

В зависимости от того, какая кнопка была нажата, функция возвращает один из результатов, представленных в таблице 9.

Таблица 9 – Возвращаемое значение

Значение	Кнопка
mrOk	«ОК»
mrCancel	«Отмена»
mrYes	«Да»
mrNo	«Нет»
mrRetry	«Повтор»
mrAbort	«Прервать»
mrIgnore	«Пропустить»

!!! Стоит обратить внимание, что строки в данном случае имеют тип **PChar**, а не **String**. Для константных значений, которые сразу указываются в кавычках внутри функции, это не имеет значения. Но если мы захотим использовать некоторую переменную **S** типа **String** внутри функции, то ее нужно будет записывать как **PChar(S)**, т.е. использовать преобразование типов.

Например, для отображения окна (рис. 3.12) с предложением заменить файл необходимо выполнить следующий код:

```
if Application.MessageBox('Заменить файл?', 'Файл существует',
                          MB_YESNO + MB_ICONQUESTION +
                          MB_DEFBUTTON2) = mrYes then
begin
  //Выполнить сохранение файла
end;
```

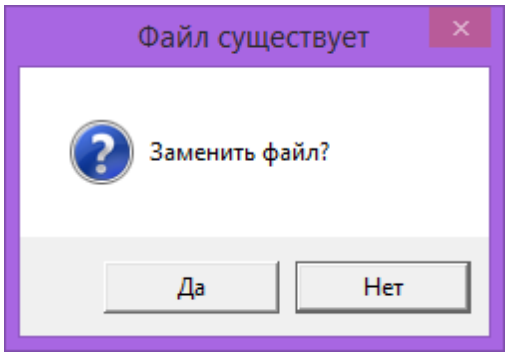


Рисунок 3.12 – Диалог замены файла

Или, например, для отображения окна (рис. 3.13) с предложением сохранить документ при закрытии программы необходимо выполнить следующий код:

```
R := Application.MessageBox('Сохранить изменения в документе?',
                            'Документ изменен',
                            MB_YESNOCANCEL + MB_ICONQUESTION);
```

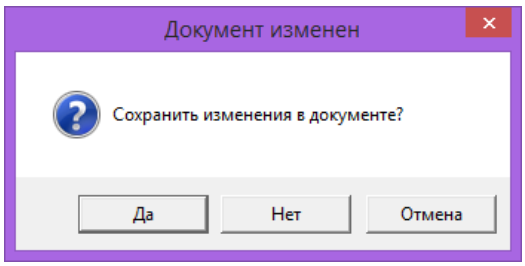


Рисунок 3.13 – Диалог сохранения изменений

Данное окно содержит три кнопки «Да», «Нет», «Отмена» (**MB_YESNOCANCEL**). Результат с выбранной пользователем кнопкой сохраняется в переменную **R**, типа **Integer**.

Пример использования данного окна, может быть следующим:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var R: Integer;
begin
  //Если файл был изменен
  if RichEdit1.Modified then
  begin
    //Отображаем диалоговое окно
    R := Application.MessageBox('Сохранить изменения в документе',
                                'Документ изменен',
                                MB_YESNOCANCEL + MB_ICONQUESTION);
    //Отменяем закрытие программы
    if R = mrCancel then CanClose := False;
    //Сохраняем файл
    if R = mrYes then FileSave1.Execute;
  end;
end;
```

!!! Этот пример с **Application.MessageBox** стоит запомнить, т.к. он далее понадобится нам при доработке «Текстового редактора» (раздел **3.6**).

3.6. РАБОТА № 2 (часть 2)

«Разработка текстового редактора»

Дополнить предыдущую часть работы следующими функциями (на этот раз с написанием кода):

- отображение в строке состояния (**StatusBar**) текущих координат каретки (номер строки и номер символа в строке);
- пункт меню «О программе», отображающий окно с именем автора, годом разработки и прочей информацией о программе;
- отдельная кнопка для выделения текста красным шрифтом;
- диалог выбора цвета текста;
- диалог выбора шрифта;
- написать код для действий «Открыть» и «Сохранить как»;
- при закрытии программы отображать окно с предложением «Сохранить изменения». Данное окно отображается только если текст был изменен (**Modified** = True). Диалог имеет три кнопки: «Да», «Нет» и «Отмена». При нажатии кнопки «Отмена» закрытие окна программы не происходит;

!!! Код для реализации перечисленных выше функций уже был рассмотрен ранее в разделах «3.2. События визуальных компонентов», «3.3. События невизуальных компонентов» и «3.5. Диалоговые окна Application.MessageBox».

- отображать надпись «Изменен» (**Modified**) в строке состояния (**StatusBar**) в случае, если текст документа был изменен;
- дополнить код для действий «Открыть» и «Сохранить как» сбрасыванием значения **Modified**;
- написать код для создания нового документа (сбрасывать **Modified**);
- написать код для печати документа;

!!! Примеры кода для работы с **Modified**, а также код для печати текстовых документов, будут рассмотрены далее в разделе «4.4. Свойства и методы полей Edit, Memo и RichEdit».

- *отдельная кнопка выделения текста желтым маркером, т.е. желтым фоном (не обязательная функция, выполняется по желанию).

!!! В данной части работы **не** обязательно:

- изменять цвет всплывающих подсказок (**HintColor**);
- настраивать диалоги открытия и сохранения файла;
- спрашивать у пользователя о возможности заменить существующий файл при сохранении;
- реализовывать кнопку «Сохранить» (без «как»). Хотя сама кнопка и должна присутствовать на панели (и в меню), но достаточно чтобы она была «серой»;
- отображать название открытого документа в заголовке окна;
- предлагать сохранить изменения при создании нового документа или открытии другого файла;

т.к. реализация данных функций вынесена в отбельную часть, для самостоятельного углубленного изучения.

Отчет должен содержать:

- краткое Задание (не более 1-ой стр.), *если автор делал программу с какими-либо дополнениями, изменениями и усложнениями, то это должно быть отражено в задании;*
- скриншоты получившейся программы;
- скриншоты Главного меню и Всплывающего меню;
- скриншоты всех окон и диалогов, разработанных в данной работе («О программе», «Сохранить изменения»);
- скриншот одного из окон «Сохранить» или «Открыть»;
- список функций, выполняемых программой;
- код главной программы (с комментариями**!**), а также код всех модулей, при их наличии;
- пример тестирования программы, т.е. скриншот с результатами ввода текста с различным форматированием;
- титульный лист, номера страниц, оглавление, список использованной литературы (включая Интернет-ресурсы и данную методичку), выводы/заключение и т.п.

4. КОМПОНЕНТЫ DELPHI, ИХ СВОЙСТВА И МЕТОДЫ

4.1. Обзор визуальных компонентов Delphi

Визуальными компонентами являются метки **Label** и поля ввода **Edit** (рис. 4.1), кнопки **Button** и различные «галочки» (рис. 4.2), компоненты для отображения изображений, таблиц и графиков (рис. 4.3), различные компоненты для ввода и/или отображения числовых значений (рис. 4.4), а также списки (рис. 4.5), выпадающие списки (рис. 4.6), панели (рис. 4.7) и др. *Некоторые компоненты попали сразу на несколько рисунков, т.к. относятся сразу к нескольким группам.*

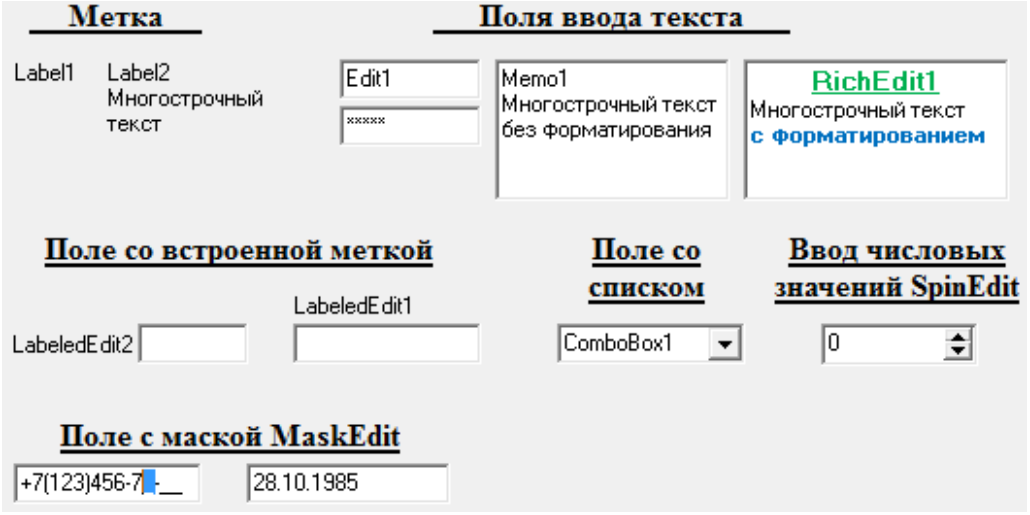


Рисунок 4.1 – Метки и поля ввода

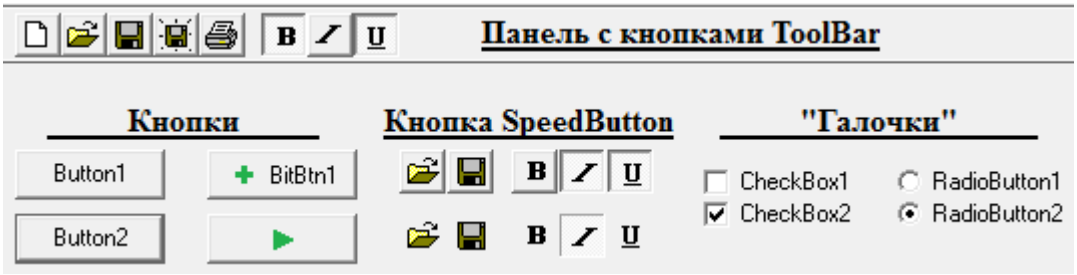


Рисунок 4.2 – Кнопки и «галочки»

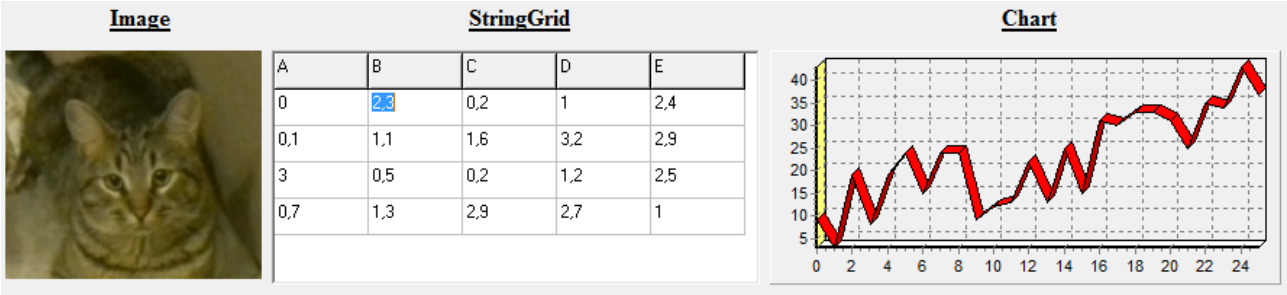


Рисунок 4.3 – Изображения, таблицы и графики

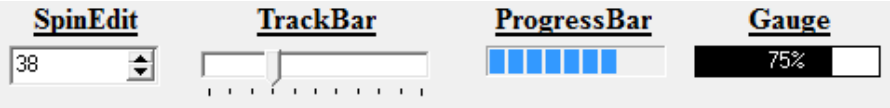


Рисунок 4.4 – Ввод и отображение числовых значений

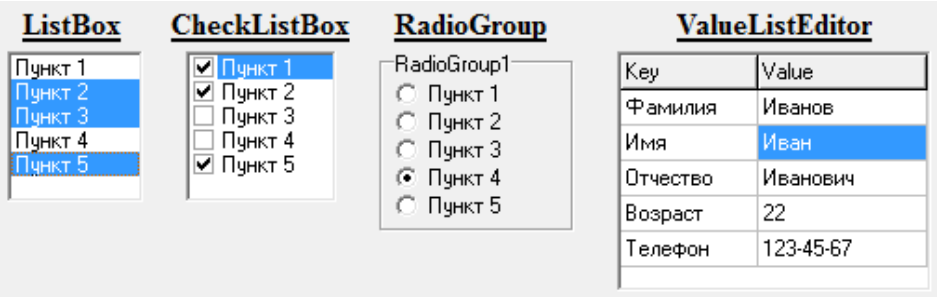


Рисунок 4.5 – Списки

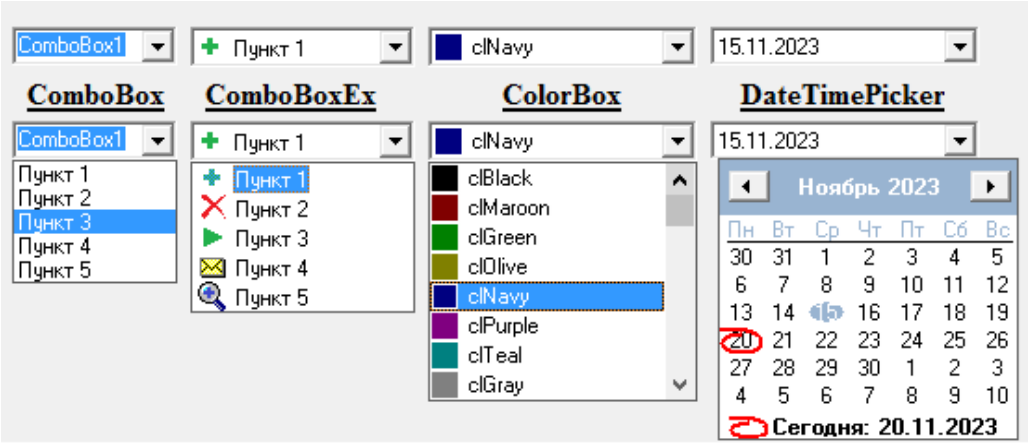


Рисунок 4.6 – Выпадающие списки

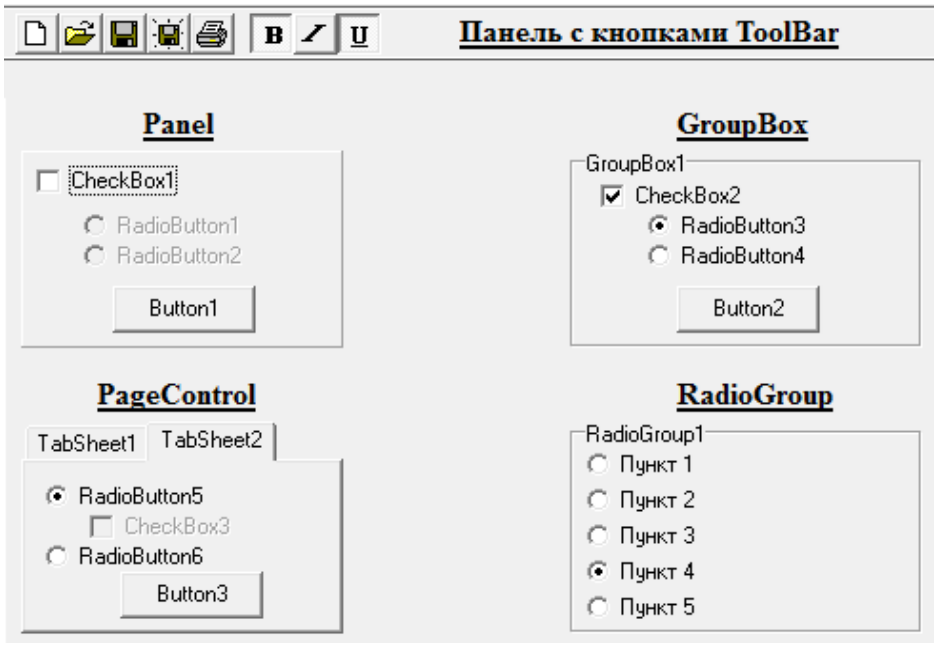


Рисунок 4.7 – Панели

Основные визуальные компоненты **Delphi** перечислены в таблице 10. Также в таблице приведены основные свойства для каждого из компонентов. При этом здесь не упоминаются общие для всех компонентов свойства, такие как **Enabled**, **Visible**, **Name**, **Left**, **Top**, **Width**, **Height**, **Align**, **PopupMenu**, **Hint**, **ShowHint**, **Cursor** и т.п. *Свойства, перечисленные вначале таблицы, могут не повторяться далее для других компонентов, т.к. работают аналогично.*

!!! Рекомендуется по мере прочтения таблицы сразу добавлять указанные компоненты на форму и тестировать их при разных значениях описанных свойств.

Таблица 10 – Основные визуальные компоненты

Компоненты	Описание
Вкладка «Standard»	
 **Label**	«Этикетка», «бирка», «пометка» или «метка». Текст «этикетки» задается через свойство **Caption**. Текст может быть многострочным, для этого необходимо задать свойство **WordWrap** := True *(переносить по словам).* Можно изменить цвет текста **Font.Color** и размер шрифта **Font.Size**. Можно сделать текст жирным **Font.Style** := [fsBold] или курсивным **Font.Style** := [fsItalic].
 **Edit**	Поле для ввода текста. Основное свойство компонента – **Text**, типа **String**. Можно ограничить количество символов, которое возможно ввести, например, **MaxLength** := 3; Может использоваться для ввода паролей, для этого нужно задать свойство **PasswordChar** := '*'. Для возврата к отображению всех символов необходимо задать **PasswordChar** := #0. Можно сделать поле доступным только для чтения (**ReadOnly** := True), тогда нельзя будет вводить текст, но можно его просматривать. *В этом случае Edit станет похож на Label, но в отличие от Label, текст в нем по-прежнему можно будет выделять и копировать (**Ctrl+C**).* Можно изменить цвет фона Edit, задав для него, например, **Color** := clBtnFace *(что означает серый цвет, такой же, как цвет фона Формы).* Или изменить цвет текста **Font.Color** и размер шрифта **Font.Size**. Можно выделить текст жирным или курсивом через свойство **Font.Style**. Можно скрыть рамку вокруг поля, задав свойство **BorderStyle** := bsNone. *Тогда он станет совсем похож на Label.* Можно сделать отображение всех букв большими (ПРОПИСНЫМИ) **CharCase** := ecUpperCase, или маленькими (строчными) **CharCase** := ecLowerCase. К полю Edit, как и к большинству других полей ввода, уже привязано системное всплывающее меню, которое

Продолжение табл. 10

Компоненты	Описание
	появляется при нажатии правой клавиши мышки. Оно содержит пункты «Копировать», «Вставить», «Выделить все» и т.д. Можно вместо него привязать свое всплывающее меню через свойство **PopupMenu**. *Для полного отключения всплывающего меню необходимо привязать пустой компонент* ***TPopupMenu***
 **Memo**	Многострочный текст. Основное свойство компонента – **Lines**, типа TStrings *(т.е. многострочный текст)*. Для этого компонента доступны такие свойства как: **WordWrap** – переносить по словам; **ReadOnly** – только для чтения; **Color**, **Font.Color**, **Font.Size** – цвет фона, цвет текста и размер шрифта; **BorderStyle** – рамка вокруг компонента; **MaxLength** – максимальное количество вводимых символов *(причем каждые* ***Enter*** *считается как два символа)*; **ScrollBars** – включение и выключение полос прокрутки *(вертикальной – справа, и горизонтальной – снизу)*.
 **Button**	Кнопка. Надпись на кнопке задается через свойство **Caption**. Ее можно сделать многострочной через свойство **WordWrap**. Для кнопки можно изменять размер шрифта **Font.Size** или сделать текст жирным или курсивом через свойство **Font.Style**. **Но** цвет фона **Color** или цвет текста **Font.Color** изменить нельзя. У кнопок есть особое свойство **Default** типа Boolean. Если его включить, то кнопка будет выделена более темной рамкой и станет «кнопкой по умолчанию». Это означает, что при нажатии клавиши **Enter** на любом поле ввода (таком как Edit), сработает эта кнопка (точнее ее событие **OnClick**). Аналогичное произойдет при нажатии **Enter** для списка и многих других компонентов. *Но это не работает для многострочных полей Memo и RichEdit, т.к. у них нажатие Enter означает переход на новую строку.* Только для одной кнопки можно задать **Default** := True.
 **CheckBox**	«Галочка», «Флажок», «Чекбокс». Основное свойство компонента – **Checked**, типа **Boolean**. Надпись на компоненте задается через свойство **Caption**. Ее можно сделать многострочной через свойство **WordWrap**.
 **RadioButton**	«Радиокнопка». Основное свойство компонента – **Checked**, типа **Boolean**. Надпись задается через **Caption**, для которого можно задать **WordWrap**.

Продолжение табл. 10

Компоненты	Описание
	Радиокнопка очень похожа на CheckBox, но предназначена только для работы **в группе** из нескольких **RadioButton** *(не менее двух)*. Все радиокнопки, помещенные на форму, автоматически объединяются в одну группу. Для того чтобы образовать отдельную группу радиокнопок, необходимо использовать панели (Panel, GroupBox и т.п.).
 **ListBox**	Список. Список вводится как многострочный текст через свойство **Items**: TStrings. Каждая строка исходного текста будет отдельным элементом списка. По списку можно перемещаться клавишами «Вверх» и «Вниз» («↑», «↓») на клавиатуре. Если задать свойство **MultiSelect** := True, то можно выделять не один, а сразу несколько пунктов списка. Выделять несколько пунктов списка **подряд** можно удерживая кнопку **Shift**. Чтобы выделить пункты **не** подряд, необходимо удерживать **Ctrl.**
 **ComboBox**	Выпадающий список (он же «Поле со списком»). Список вводится как многострочный текст через свойство **Items**: TStrings. Т.к. **ComboBox** является не только списком, но и полем для ввода (как Edit), то его основным свойством, как и у других полей, является **Text**. В это поле можно вводить любой текст, а не только указанный в списке. **ComboBox** обладает и некоторыми другими свойствами полей Edit, такими как: **MaxLength**, или **CharCase**.Если задать свойство **AutoDropDown** := True, то при вводе текста с клавиатуры, список будет автоматически раскрываться и выбирать первый из пунктов, подходящий под введенный текст. По списку можно перемещаться клавишами «Вверх» и «Вниз» («↑», «↓») на клавиатуре. Свойство компонента **ItemIndex** указывает номер выбранного пункта списка (начиная с нуля). Если ни один пункт не выбрана, то **ItemIndex** = -1. Можно задать свойство **Style** := csDropDownList, тогда можно будет выбирать только значения из списка, и нельзя вводить произвольный текст. В этом случае основным свойством может выступать уже **ItemIndex**, а **Text** можно вообще не использовать. Ограничить количество строк, отображаемых в выпадающем списке, можно при помощи свойства **DropDownCount.**
 **Panel**	Панель. Все панели предназначены для **визуального** объединения компонентов в одну группу. В случае с Радиокнопками, это объединение еще и **функциональное**.

Продолжение табл. 10

Компоненты	Описание
	Панели являются «**контейнерами**», т.е. в них можно разместить любые другие визуальные компоненты (в т.ч. другие панели). Форма TForm также является контейнером. Можно отключить рамку вокруг компонента Panel, задав свойство **BevelOuter** := bvNone. Другими свойствами, отвечающими за различные варианты отображения рамки, являются: **BevelInner**, **BevelWidth** и **BorderStyle**. Для этой панели обязательно нужно задавать **Caption** := ' ', т.к. надпись посередине панели совершенно не нужна.
 **GroupBox**	Панель с заголовком и рамкой. Используется как Контейнер. Заголовок задается через **Caption.**
 **RadioGroup**	Группа радиокнопок, Панель с радиокнопками. Внешне совпадает с GroupBox (т.е. с панелью с заголовком и рамкой), но, на самом деле, **не является контейнером** (в нее нельзя помещать другие компоненты). Является видом Списка (как ListBox или ComboBox) и имеет свойство **Items**: TStrings с многострочным текстом, каждая строка которого добавляет отдельную радиокнопку на данную «панель». Можно задать количество колонок при помощи свойства **Columns**. Основное свойство компонента – **ItemIndex**, указывающее номер выбранной радиокнопки (начиная с нуля). Если ни одна радиокнопка не выбрана, то **ItemIndex** = -1. *Стоит напомнить, что только одна радиокнопка в группе может быть выбрана.*
Вкладка «Additional»	
 **BitBtn**	Вариант кнопки с изображением. Изображение задается через свойство **Glyph**. В остальном данная кнопка аналогична обычной кнопке TButton, в т.ч. для нее можно задать свойство **Default.**
 **SpeedButton**	Вариант кнопки с изображением. Изображение также задается через свойство **Glyph**, но данная кнопка сильно отличается от обычной кнопки TButton, в т.ч. у нее **нет** свойства **Default**. Зато она позволяет зажать кнопку в нижнем положении (свойство **Down**), а также объединять кнопки в группы при помощи свойства **GroupIndex**. Если **GroupIndex** = 0, то группировка отключена и не удастся зажать кнопку. В группе с одинаковым номером может быть зажата только одна кнопка. Если мы хотим, чтобы все кнопки в группе могли быть отжаты (т.е. быть в верхнем положении), то необходимо установить **AllowAllUp** := True. У этой кнопки все еще имеется свойство **Caption**, поэтому мы, как и прежде, можем задать надпись на кнопке.

Продолжение табл. 10

Компоненты	Описание
	Свойство **Flat** := True – позволяет скрыть рамку вокруг кнопки.
 **MaskEdit**	Поле ввода с маской. Предназначено для ввода номеров телефонов, даты и времени, почтовых индексов и др. значений, для которых заранее задан определенных формат записи. Маска вводится через свойство **EditMask**. Если дважды щелкнуть по этому значению (или нажать в нем кнопку с многоточием «…»), то откроется Мастер, в котором можно выбрать один из готовых шаблонов. *К сожалению, имеющиеся варианты плохо подходят для России, поэтому их придется немного подправить под себя.*
 **StringGrid**	Таблица. Количество строк и столбцов задается свойствами **RowCount** и **ColCount**.По умолчанию для таблицы отключен ввод значений в ячейки с клавиатуры. Для его включения необходимо в свойстве **Options** установить **goEditing** := True.
 **Image**	Изображение. Основное свойство – **Picture**. В него можно загрузить изображение в формате BMP, JPEG и др. Если размеры загруженного изображения отличаются от размеров самого Image, то необходимо установить свойство **Proportional** := True, тогда изображение будет промасштабировано под размер компонента, но с сохранением пропорций исходного изображения. *Если пропорции не важны, то вместо **Proportional** можно установить свойство **Stretch** := True.*
 **Bevel**	Рисует рамки и линии. Свойство **Shape** позволяет выбрать тип рамки или тип линии (вертикальная или горизонтальная) Не является панелью и не может быть контейнером для других компонентов. У этого компонента полностью отсутствуют какие-либо события (нет даже OnClick). Но для данной области можно задать свои **Cursor** и **Hint.**
 **CheckListBox**	Список «галочек». Список вводится как многострочный текст через свойство **Items**: TStrings. Каждая строка исходного текста будет отдельным элементом списка. Список будет автоматически отсортирован по возрастанию, если включить **Sorted** := True. По списку можно перемещаться клавишами «Вверх» и «Вниз» («↑», «↓») на клавиатуре, а галочки включать и выключать нажатием клавиши «Пробел». Каждая галочка включается независимо, можно включить любое их количество (в т.ч. можно выключить все). Можно задать количество колонок при помощи свойства **Columns**.

Продолжение табл. 10

Компоненты	Описание
	Можно скрыть рамку вокруг поля, задав свойство **BorderStyle** := bsNone или изменить цвет фона **Color** (например, на тот же цвет, что имеет окно).
 **Splitter**	Разделитель. Позволяет изменять размеры панелей и компонентов пользователем во время работы программы (в **Run-time**). Применяется к компонентам, для которых установлено значение **Align**. Для самого **Splitter**, также нужно задать соответствующий **Align**. Может быть как горизонтальным, так и вертикальным.
 **ValueListEditor**	Список значений. Таблица из двух колонок «Key» («Ключ», «Имя») и «Value» («Значение»). Основное свойство – **Strings** типа TStrings. Через него задается список «Ключей» и их «Значения» по умолчанию. Например, можно создать таблицу со строками: Фамилия, Имя, Отчество и Возраст, а пользователь впишет свои значения. Через свойство **TitleCaptions** можно изменить заголовки колонок в шапке таблицы (изначально это «Key» и «Value»). **ValueListEditor** имеет множество настроек, осуществляемых через свойства **Options**, **KeyOptions** и **DisplayOptions**. Во время работы программы, пользователь может изменять только «Значения», но не имена «Ключей». Кроме того, пользователь не может добавлять новые строки или удалять существующие. Но через свойство **KeyOptions** можно разрешить изменение, добавление и удаление «Ключей». Удаление строки («Ключа») в этом случае производится нажатием **Ctrl+Del**. Для добавления новой строки необходимо переместиться клавишей клавиатуры «Вниз» («↓») ниже последней строки, чтобы появилась новая строка.
 **LabeledEdit**	Вариант поля Edit со встроенной надписью Label. Текст задается, как и обычно для Edit, через свойство **Text**, а надпись через **EditLabel.Caption**. Подпись может располагаться сверху, снизу, слева или справа, что задается свойством **LabelPosition**.

Продолжение табл. 10

Компоненты	Описание
 **ColorBox**	Выпадающий список с цветами. Главное свойство – **Selected**, показывающее, какой цвет выбран в списке. Настроить список доступных для выбора цветов можно через свойство **Style**. В нем рекомендуется отключать системные цвета **cbSystemColors**, а также включить **cbCustomColor**, который позволит выбирать не только фиксированные цвета из списка, но и вызвать Диалог для выбора произвольного цвета.
 **Chart**	График.
Вкладка «Win32»	
 **PageControl**	Многостраничная панель. Позволяет экономить пространство окна, распределяя информацию по вкладкам. Часто используются для окон с настройками. Каждая вкладка (страница), фактически является отдельной панелью типа **TTabSheet** *(и отдельным независимым контейнером для размещения других компонентов)*. Для добавления новой вкладки необходимо нажать на **PageControl** правой клавишей мышки и выбрать пункт меню «**New Page**». Надпись на вкладке задается через свойство **Caption**. Вкладки могут иметь не только надписи, но и иконки, выбираемые через **ImageIndex**. При этом **PageControl** должен быть связан с **ImageList** (в который заранее нужно загрузить изображения).
 **RichEdit**	Многострочный текст с форматированием. Позволяет выделить текст жирным или курсивом, изменить цвет или размер шрифта и т.п. Основное свойство компонента – **Lines**, типа **TStrings**. *На этапе проектирования можно указать только текст без форматирования**!** Форматирование можно применить во время работы программы. Можно вставлять форматированный текст из **Word** или копировать его в **Word**. Также форматированный текст можно загрузить из файла в формате *.**rtf**.* В остальном практически полностью совпадает с Memo.

Продолжение табл. 10

Компоненты	Описание
 **TrackBar**	Ползунок. Может перемещаться как мышкой, так и клавишами клавиатуры «Влево», «Вправо», «Вверх», «Вниз» («←», «→», «↑», «↓»). При помощи свойства **Orientation** можно сделать его горизонтальным или вертикальным. Основное свойство компонента – **Position** типа Integer. Также можно задать значения границ **Min** и **Max** (по умолчанию от 0 до 10).
 **ProgressBar**	Отображение процента выполнения. При помощи свойства **Orientation** можно сделать его горизонтальным или вертикальным. Основное свойство компонента – **Position** типа Integer. Также можно задать значения границ **Min** и **Max** (по умолчанию от 0 до 100).
 **MonthCalendar**  **DateTimePicker**	Календарь и выпадающий «список» с календарем.
 **TreeView**	Дерево. Позволяет отображать иерархические данные в виде дерева (например, такие, как структура каталогов на диске ПК). Отдельные ветви дерева можно сворачивать и разворачивать.
 **StatusBar**	Строка состояния (она же Панель состояния), расположенная внизу окна.
 **ToolBar**	Панель с кнопками, обычно расположенная сверху. Для добавления новой кнопки необходимо нажать на **ToolBar** правой клавишей мышки и выбрать пункт меню «**New Button**». Кроме того, можно добавить разделители между кнопками, выбрав «**New Separator**». Изображение на кнопках задаются через свойство **ImageIndex**. При этом **ToolBar** должен быть связан с **ImageList** (в который заранее нужно загрузить изображения). Свойство **Flat** := True – позволяет скрыть рамки вокруг кнопок (как это было у **SpeedButton**). Если для кнопки задать **Style** := **tbsCheck**, то эту кнопку можно будет переключать между нижним (зажатым) и верхним (отжатым) положением. Кнопка будет находится в нижнем положении при **Down** := True, а в верхнем положении при **Down** := False. Для объединения кнопок в группы необходимо задать для них **Grouped** := True. Все кнопки (с включенным **Grouped**),

Окончание табл. 10

Компоненты	Описание
	идущие подряд, образуют единую группу. Для разделения кнопок на несколько групп, между ними должна располагаться кнопка с отключенной группировкой, разделитель (**Separator**), или любой иной компонент (выпадающий список, чекбокс, радиокнопка и т.п.). В группе может быть одновременно зажата только одна кнопка. Если мы хотим, чтобы все кнопки в группе могли быть отжаты (т.е. быть в верхнем положении), то необходимо установить **AllowAllUp** := True. **ToolBar** является «Контейнером», поэтому на эту панель можно добавить не только стандартные кнопки и разделители, но и любые другие визуальные компоненты (например, выпадающие списки, чекбоксы и радиокнопки).
 **ComboBoxEx**	Вариант выпадающего списка, позволяющий добавлять к пунктам списка иконки (из ImageList).
Вкладка «Samples»	
 **Gauge**	Отображение процента выполнения. При помощи свойства **Kind** можно сделать его горизонтальным, вертикальным, круглым или полукруглым. Основное свойство компонента – **Progress** типа Integer. Также можно задать значения границ **MinValue** и **MaxValue** (по умолчанию от 0 до 100).
 **ColorGrid**	Выбор цвета. Позволяет выбирать только из 16-ти фиксированных цветов. При этом можно выбирать сразу два цвета, один правой кнопкой мышки, другой левой.
 **SpinEdit**	Поле ввода числовых значений. Основное свойство компонента – **Value**, типа **Integer**. Можно ограничить диапазон вводимых чисел при полощи свойств **MaxValue** и **MinValue**. Содержит дополнительные кнопки увеличения и уменьшения значения. Используя свойство **Increment**, можно задать шаг изменения значений при помощи этих кнопок.

Пример бурной деятельности

Необходимо каждую секунду генерировать случайное число, отображаемое в **Edit**, а также увеличивать значение **ProgressBar** на **2 %**. При достижении **100 %,** остановить генерацию. Внешний вид программы представлен на рисунке 4.8. *Вместо* ***ProgressBar*** *можно также использовать компонент* ***Gauge***.

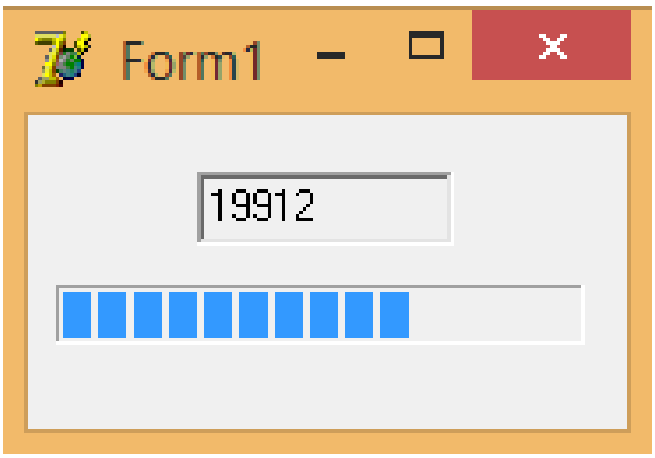


Рисунок 4.8 – Пример бурной деятельности

Код для данной программы будет следующим:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Randomize;
  Edit1.Text := IntToStr(Random(100000));

  ProgressBar1.Position := ProgressBar1.Position + 2;
  if ProgressBar1.Position >= 100 then Timer1.Enabled := False;
end;
```

4.2. Графический интерфейс пользователя

GUI (**G**raphical **U**ser **I**nterface, Графический интерфейс пользователя) – система средств для взаимодействия пользователя с электронными устройствами, основанная на представлении всех доступных пользователю системных объектов в виде графических компонентов экрана (окон, значков, меню, кнопок, списков и т. п.).

GUI является стандартной составляющей большинства современных операционных систем, таких как: Microsoft Windows, Mac OS, Linux, Android, iOS и др.

GUI-конструктор (**GUI-редактор**, Конструктор графического пользовательского интерфейса) – является инструментарием разработки программного обеспечения, который упрощает создание графического интерфейса пользователя (**GUI**), позволяя разработчику упорядочить элементы интерфейса используя **drag-and-drop** («тащи-и-бросай») при помощи мышки. Без **GUI-конструктора** графический интерфейс пользователя приходится создавать вручную, с указанием параметров каждого элемента интерфейса в исходном коде и без визуальной обратной связи (до запуска программы). Пользовательские интерфейсы обычно программируются с помощью **событийно-ориентированной** архитектуры, поэтому **GUI-конструкторы** также упрощают создание кода, управляемого событиями.

Delphi (и **C++ Builder**) имеет в своем составе **GUI-редактор**, в котором и создаются окна будущей программы.

В противоположность Графическому интерфейсу, существует также «**Интерфейс командной строки**». **Delphi** позволяет разрабатывать и такие приложения, но, все же, основное его назначение – создание программ с Графическим интерфейсом.

4.3. Свойства и методы компонентов

Свойства компонентов

Ранее уже были рассмотрены основные свойства компонентов, как визуальных, так и некоторых невизуальных.

Свойства можно изменять при проектировании программы (в **Design-time**) через «Инспектор объектов». Их мы будем называть «**визуальными**» свойствами. **События** компонентов также являются одним из видов свойств, и их также можно задавать через «Инспектор объектов» (но необходимо перейти на вкладку «**Events**»).

Все свойства, которые можно изменять через «Инспектор объектов», можно также изменять и из кода (т.е. в **Run-time**). Например:

```
Edit1.Text := 'Привет!'; //Изменяем текст из кода
```

Кроме того, у компонентов существуют также свойства, которые нельзя задавать через «Инспектор объектов» (в **Design-time**). Их можно изменять **только из кода** программы (в **Run-time**). Например:

```
Edit1.SelStart := 3; //Изменяем положение каретки
```

Их мы будем называть «**невизуальными**» свойствами.

Описанные выше **Свойства** компонентов можно рассматривать как специальный вид «**переменных**», но не обычных переменных, объявленных как **var**, а как переменных, принадлежащих компоненту.

Методы компонентов

Кроме **Свойств**, у компонентов существуют еще и **Методы**.

Методы – это процедуры и функции, но не любые, а только принадлежащие к конкретному компоненту. Эти процедуры и функции не используются в **Design-time** (и никогда не отображаются в «Инспекторе объектов»), их можно использовать только в **коде** программы, например:

```
Edit1.Clear; //Процедура, очищающая содержимое поля Edit
```

4.4. Свойства и методы полей Edit, Memo и RichEdit

Основные невизуальные свойства и методы полей для ввода текста (**Edit**, **Memo**, **RichEdit**) представлены в таблице 11. *В данном разделе мы не будем повторно рассматривать визуальные свойства компонентов, которые отображаются в Инспекторе объектов.*

Таблица 11 – Основные невизуальные свойства и методы полей ввода

Свойства и методы	Описание
procedure **Clear**; procedure **ClearSelection**;	Очищает все содержимое поля, или очищает выделенный текст.
procedure **CopyToClipboard**; procedure **CutToClipboard**; procedure **PasteFromClipboard**;	Копирует или вырезает в буфер обмена выделенный текст. Или вставляет текст из буфера обмена. Вставка происходит в ту позицию, где находится каретка. *Для этих и некоторых других методов в **Delphi** уже созданы стандартные **Action**. Они используют эти методы, поэтому пользователю их уже можно не писать. Но стоит помнить, что стандартные **Action** применяются не к конкретному компоненту, а к тому компоненту, который находится в фокусе.*
procedure **SelectAll**;	Выделяет весь текст в поле.
procedure **Undo**;	Отменяет последнее действие. *Повторное применение отменяет отмену, т.е. возвращает текст, который был до применения первой отмены.*
SelStart: Integer; **SelLength**: Integer; **SelText**: String;	Положение каретки (или начало выделения текста), длина выделенного текста (количество выделенных символов) и текст выделенного фрагмента. *Стоит обратить внимание, что **Enter** (т.е. переход на новую строку) занимает сразу **2** символа.*
function **Focused**: Boolean; procedure **SetFocus**;	**Focused** – возвращает True, если фокус находится на данном компоненте; **SetFocus** – устанавливает фокус на данный компонент.
Modified: Boolean;	«Модифицирован», «Изменен». Автоматически устанавливается в **True** при вводе текста пользователем. *При изменении текста из кода,* **Modified** *не изменяется для Edit и Memo. Но для RichEdit **Modified** изменяется и из кода (например, при вводе RichEdit1.Lines.Text := '123').* Сбрасывается в **False** только вручную (из кода). Может использоваться для того, чтобы определить, нужно ли сохранять текст при закрытии программы (был ли документ изменен).

Будем отображать состояние **Modified** в панели состояния. Обновлять значение будем по таймеру:

```
{-------------------------- Обновление по таймеру ---------------------------}
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if RichEdit1.Modified then StatusBar1.Panels[1].Text := 'Modified'
                        else StatusBar1.Panels[1].Text := '';
end;
```

При этом для таймера необходимо задать время **0,1-0,2** секунды.

Значение **Modified** нужно сбрасывать вручную. Тогда для сохранения и загрузки файла правильно будет писать:

```
//Сохранить в файл
RichEdit1.Lines.SaveToFile(FileSaveAs1.Dialog.FileName);
RichEdit1.Modified := False;

//Загрузить из файла
RichEdit1.Lines.LoadFromFile(FileOpen1.Dialog.FileName);
RichEdit1.Modified := False;
```

Для создания нового документа используется следующий код:

```
//Создание нового (чистого) документа
RichEdit1.Clear;
RichEdit1.Modified := False;
```

!!! Эти примеры с **Modified** и **Clear** стоит запомнить, т.к. они понадобятся нам при доработке «Текстового редактора» (раздел **3.6**).

Многострочные поля

Многострочные поля **Memo** и **RichEdit** включают все перечисленные выше свойства и методы, а также содержат некоторые дополнительные (табл. 12), которых нет у обычных полей **Edit**.

Таблица 12 – Основные невизуальные свойства многострочных полей

Свойства	Описание
CaretPos: TPoint; *где* **TPoint** = record X: Longint; Y: Longint; end;	Положение каретки. Где **CaretPos.Y** – номер строки (начиная с нуля); **CaretPos.X** – номер символа в этой строке (начиная с нуля)
Lines: TStrings;	Многострочный текст. Содержит следующие собственные свойства и методы: **Lines.Text** – текст; **Lines.Count** – количество строк текста; **Lines.Add** – добавляет новую строку в конец текста; **Lines.LoadFromFile**, **Lines.SaveToFile** – загружает текст из файла, или сохраняет текст в файл. Для **Memo** это обычный текст в формате ***.txt**. Для **RichEdit** это форматированный текст ***.rtf**; **Lines.Delete**, **Lines.Insert**, **Lines.Move**, **Lines.Exchange** – удаление, добавление и перемещение строк; и др.

Например, создадим новый текстовый документ, состоящий из трех строк. Если в поле **Memo** раньше был другой текст, то он будет потерян:

```
Memo1.Clear;
Memo1.Lines.Add('Строка 1');
Memo1.Lines.Add('Строка 2');
Memo1.Lines.Add('Строка 3');
```

Поля RichEdit

Для форматированного текста **RichEdit** имеются дополнительные свойства и методы (табл. 13).

Таблица 13 – Основные невизуальные свойства и методы RichEdit

Свойства и методы	Описание
SelAttributes: TTextAttributes;	Задает формат для выделенного текста. Например: //Красный жирный текст RichEdit1.**SelAttributes**.Color := clRed; RichEdit1.**SelAttributes**.Style := [fsBold];

Paragraph: TParaAttributes;	Задает формат для параграфа, на котором находится каретка. *Можно задавать формат сразу для нескольких параграфов, для этого необходимо выделить текст в этих параграфах.*
	Например:
	//Выравнивание по центру
	RichEdit1.**Paragraph**.Alignment := taCenter;
	//Отступ красной строки в 10 пикселей
	RichEdit1.**Paragraph**.FirstIndent := 10;

procedure **Print**(Caption: String);	Выводит документ на печать. Печать осуществляется на принтере, который выбран через **PrintDialog** (или через Action, содержащий этот диалог). **Caption** указывает название документа, которое будет отображаться в «Очереди печати». При печати на виртуальный **PDF**-принтер, это будет еще и имя сохраняемого файла

Для печати форматированного текста, используется следующий код:

```
{----------------------------- Печать документа -----------------------------}
procedure TForm1.PrintDlg1Accept(Sender: TObject);
var Caption: String;
begin
  Caption := 'Новый документ';
  RichEdit1.Print(Caption);
end;
```

!!! Этот пример с **Print** стоит запомнить, т.к. он понадобятся нам при доработке «Текстового редактора» (раздел **3.6**).

Для типа **TTextAttributes** и, соответственно, для свойства **SelAttributes**, имеются следующие собственные свойства (табл. 14).

Таблица 14 – Основные свойства TTextAttributes

Свойства и методы	Описание
Size: Integer;	Размер шрифта
Color: TColor;	Цвет текста
Name: TFontName;	Имя шрифта
Style: TFontStyles;	Позволяет сделать текст жирным, курсивным, подчеркнутым или зачеркнутым. Например: RichEdit1.SelAttributes.Style := [**fsBold**]; или: RichEdit1.SelAttributes.Style := [**fsBold**, **fsItalic**];

Для типа **TParaAttributes** и, соответственно, для свойства **Paragraph**, имеются следующие собственные свойства (табл. 15).

Таблица 15 – Основные свойства TParaAttributes

Свойства и методы	Описание
Alignment: TAlignment; *где* **TAlignment** = (taLeftJustify, taRightJustify, taCenter);	Выравнивание текста по левому краю, по правому краю или по центру
Numbering: TNumberingStyle; *где* **TNumberingStyle** = (nsNone, nsBullet)	Вставка маркеров списка
FirstIndent: Longint;	Отступ красной строки (в пикселях)
LeftIndent: Longint; **RightIndent**: Longint;	Отступы от левого и правого краев (в пикселях)

4.5. Методы Execute

Execute переводится как «Выполнить». Методы с этим именем есть (независимо друг от друга) у нескольких компонентов.

Выполнить Действие

Обычно, **Action** (Действия) выполняют свой код автоматически, в ответ на действие пользователя (нажатие кнопки на панели, выбор пункта меню), но мы всегда можем выполнить Действие принудительно из кода (имитировать нажатие кнопки или выбор пункта меню), например:

```
Action1.Execute;
```

Execute объявлен как **функция**, возвращающая логическое значение:

```
function Execute: Boolean;
```

но часто она используется только как **процедура**, т.е. без получения результата выполнения. Это связано с тем, что при нормальной работе эта функция всегда возвращает **True**. *А **False** будет получен только если для Action не присвоен обработчик **OnExecute**.*

!!! Практический пример использования **Execute** для Действия уже был рассмотрен ранее, в разделе **3.5**.

Выполнить Диалог

Для Диалогов, **Execute** также объявлен как **функция**, возвращающая логическое значение:

```
function Execute: Boolean;
```

Например, для запуска диалога сохранения в файл, можно выполнить следующий код:

```
if SaveDialog1.Execute then
begin
  //Сохранить в файл
  RichEdit1.Lines.SaveToFile(SaveDialog1.FileName);
end;
```

При этом **Execute** возвращает результат, который мы проверяем в условии **if**. Если пользователь нажмет в диалоговом окне кнопку «Отмена», то **Execute** вернет **False** и код сохранения в файл не будет выполнен.

!!! При выполнении кода, содержащего **Execute**, программа пользователя «замирает» на время, пока отображается диалог, и будет продолжена с этого места только когда Диалог будет закрыт. *Но код пользователя в других обработчиках событий продолжит выполняться, например, будут работать таймеры.*

4.6. *Системные компоненты Application, Screen и Mouse

В Delphi имеется несколько системных Компонентов. Для них уже созданы глобальные переменные с одним экземпляром каждого Компонента:

Application: TApplication;
Screen: TScreen;
Mouse: TMouse;

!!! Данные Компоненты нельзя найти в Палитре компонентов и нельзя добавить их на форму. Но в этом и нет необходимости, они всегда существуют в программе в одном (и только в одном!) экземпляре.

С помощью этих Компонентов можно, например, узнать размеры и разрешение экрана, количество мониторов, шрифты, установленные в системе, текущее положение курсора мышки, активна ли сейчас программа или задать внешний вид курсора мышки, настроить цвет всплывающих подсказок (Hint) и время их появления, задать имя Приложения и т.п. Некоторые из свойств предназначены только для чтения, но другие можно изменить.

TApplication

Компонент **Application** [1,2] («Приложение») – позволяет получить информацию о Приложении. Его основные свойства представлены в таблице 16.

Таблица 16 – Основные свойства TApplication

Свойства	Описание
Application.**Active**: Boolean;	Активность приложения в ОС (т.е. имеет ли приложение фокус ввода). *Только одна из всех запущенных в ОС программ может иметь фокус ввода.*
Application.**ExeName**: String;	Полный путь к запущенному exe-файлу. *Чтобы получить **папку**, из которой запущена программа, можно использовать следующую функцию:* `ExtractFilePath(Application.ExeName)`
Application.**Title**: String; Application.**Icon**: TIcon;	Заголовок и иконка приложения в панели задач Windows (внизу экрана). *Задать заголовок и иконку приложения можно и через меню «Project» → «Options» → «Application», но отличие данного способа в том, что можно менять заголовок и иконку динамически.* *Иконки можно загружать из файлов, командой:* `Application.Icon.LoadFromFile(...);`
Application.**Hint**: String;	Текст текущей подсказки. Подсказка для кнопки (или пункта меню), над которым в настоящее время находится курсор мышки. *Если курсор мышки не находится над кнопкой (пунктом меню), то **Application.Hint** будет пустым.* Обычно используется совместно с **OnHint** (см. далее). *Если Hint кнопки (пункта меню) содержит символ «|», то здесь будет отображена только вторая половина этого текстового описания, а первая половина будет отображена во всплывающей подсказке.*

Application.**ShowHint**: Boolean;
Application.**HintColor**: TColor;
Application.**HintPause**: Integer = **500**;
Application.**HintHidePause**: Integer = **2500**;
Application.**HintShowPause**: Integer = **0**;
	Настройки отображения всплывающих подсказок.
	Например, можно изменить цвет подсказки (**HintColor**) с желтого на розовый:
	`//Изменить цвет всплывающих подсказок`
	`Application.HintColor := $FF80FF;`
	Задержка появления подсказки (**HintPause**), время отображения подсказки (**HintHidePause**) и задержка на переключение между подсказками (**HintShowPause**) задаются в миллисекундах (например, 2500мс = 2,5с).

TApplicationEvents

Компонент **Application** нельзя увидеть из Инспектора объектов, и, соответственно, невозможно работать с его событиями привычным нам способом. К счастью, в Палитру компонентов Delphi разработчики добавили **TApplicationEvents** [1,3], который можно перетащить на форму и дальше пользоваться его событиями как обычно.

!!! На каждой форме можно разместить свою копию Компонента. События, наступающие в **Application**, будут передаваться всем **ApplicationEvents** на всех формах.

Основные события TApplicationEvents приведены в таблице 17.

Таблица 17 – Основные события TApplicationEvents

Событие	Описание
OnActivate: TNotifyEvent; **OnDeactivate**: TNotifyEvent; **OnMinimize**: TNotifyEvent; **OnRestore**: TNotifyEvent;	Срабатывают, когда Приложение становится активным, перестает быть активным (например, пользователь перешел к другой программе), когда Приложение сворачивается или разворачивается обратно.
OnException: TExceptionEvent;	Срабатывает, когда в Приложении возникает исключение (которое больше нигде не обработано!). В отличие от перехвата исключений в конкретном месте кода (при помощи **try-except**), здесь перехватываются сразу все необработанные исключения во всех местах кода. *Может быть полезно, например, для перевода определенных сообщений об ошибках на русский, или для изменения дизайна сообщений об ошибках. Позволяет вовсе отключить определенные сообщения об ошибках или все их разом (что, конечно, не рекомендуется делать!)*
OnIdle: TIdleEvent;	Срабатывает, когда Приложение начинает простаивать [4] (от англ. Idle – «безделье», «незанятый», «простаивать»), т.е. когда не выполняется ни одни из обработчиков событий, написанных программистом. **!!!** Код в OnIdle должен быть коротким по времени выполнения, иначе это приведет к «притормаживанию» всего Приложения!

Окончание табл. 17

Событие	Описание
OnHint: TNotifyEvent;	Срабатывает, когда курсор мышки начинает перемещаться над компонентом или пунктом меню, для которого задано свойство **Hint** (при этом не принципиально, включен **ShowHint** или нет). В этот момент текст текущего Hint можно получить из **Application.Hint**. Это же событие срабатывает, когда курсор мышки покидает все компоненты с Hint. В этом случае текст в **Application.Hint** будет пустым. *Обычно **OnHint** используется для вывода подробного описания для кнопки (пункта меню) в строке состояния (см. далее).*
OnShowHint: TShowHintEvent;	Срабатывает перед тем, как будет отображена всплывающая подсказка. *Здесь можно изменить текст всплывающей подсказки (включая указание «горячих клавиш»), запретить ее показ, изменить координаты, время показа или другие параметры отображения, задать свои цвета для каждой из всплывающих подсказок, сделать так, чтобы длинные подсказки отображались на экране дольше, чем короткие.*

Application.Hint и ApplicationEvents.OnHint

Для отображения в строке состояния (рис. 4.9) описания (для кнопки или пункта меню) необходимо написать следующий обработчик события **OnHint** для компонента **ApplicationEvents**:

```
{------------- Отобразить подсказку в панели состояния --------------}
procedure TForm1.ApplicationEvents1Hint(Sender: TObject);
begin
  StatusBar1.Panels[2].Text := Application.Hint;
end;
```

!!! Стоит обратить внимание, что если **Hint** кнопки (пункта меню) содержит символ «|», то здесь будет отображена только **вторая** половина этого текстового описания. *А **первая** половина будет отображена во всплывающей подсказке.*

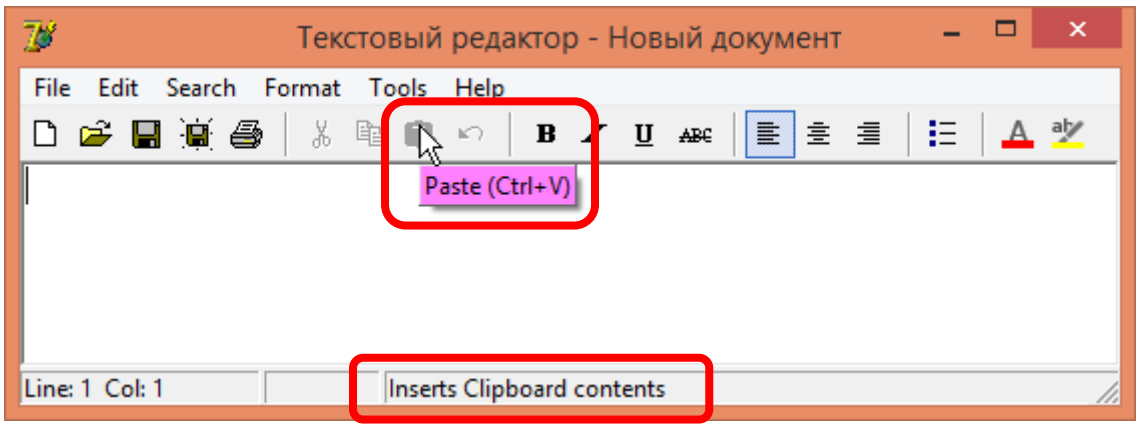


Рисунок 4.9 – Подсказка в панели состояния

!!! В студенческих работах все подсказки должны быть переведены на русский.

TScreen

Компонент Screen позволяет получить информацию о текущем состоянии экрана. Основные свойства TScreen представлены в таблице 18.

Таблица 18 – Основные свойства TScreen

Свойства	Описание
Screen.**Width**: Integer; Screen.**Height**: Integer;	Ширина и высота экрана (в пикселях)
Screen.**PixelsPerInch**: Integer;	Разрешение экрана. *Например, 96 точек на дюйм*
Screen.**WorkAreaWidth**: Integer; Screen.**WorkAreaHeight**: Integer; Screen.**WorkAreaLeft**: Integer; Screen.**WorkAreaTop**: Integer;	Размеры и положение рабочей области экрана. *Рабочая область может быть меньше размеров экрана, когда часть экрана занята панелью задач Windows или другими панелями. Например, высота меньше на 40 пикселей при отображении панели задач (и кнопки «Пуск»)*

Screen.**Cursor**: TCursor;	Курсор мышки
	Можно изменить курсор, например:
	`//Курсор ожидания завершения работы`
	`Screen.Cursor := crHourGlass;`

Screen.**ActiveForm**: TForm; Screen.**FormCount**: Integer; Screen.**Forms**[i]: TForm;	Активная форма, количество и список всех форм
Screen.**Fonts**: TStrings;	Список шрифтов, установленных в системе

****Несколько мониторов**

В системе может быть установлено несколько мониторов, получить информацию о них также можно через TScreen (табл. 19).

Таблица 19 – Свойства и методы TScreen для работы с несколькими мониторами

Свойства и методы	Описание
Screen.**MonitorCount**: Integer; Screen.**Monitors**[i].**Width**: Integer; Screen.**Monitors**[i].**Height**: Integer; Screen.**Monitors**[i].**Left**: Integer; Screen.**Monitors**[i].**Top**: Integer;	Количество мониторов, их размеры и положения. **!!!** *Количество мониторов не обновляется после запуска Приложения, т.е. все мониторы должны быть подключены до запуска программы***!**
Screen.**DesktopWidth**: Integer; Screen.**DesktopHeight**: Integer; Screen.**DesktopLeft**: Integer; Screen.**DesktopTop**: Integer;	Размеры виртуального рабочего стола, состоящего из всех подключенных мониторов. *В системе с одним монитором,* ***DesktopWidth*** *и* ***DesktopHeight*** *совпадают с* ***Width*** *и* ***Height****.* **!!!** *Настроить положение мониторов относительно друг друга, можно в Панели управления Windows на странице «Разрешение экрана»*

Screen.**MonitorFromPoint**(Point).**MonitorNum**: Integer;	Возвращает номер монитора, на котором расположена указанная точка Point, имеющая координаты X и Y.
	Например, получение номера монитора, над которым находится курсор мышки:
	`Screen.MonitorFromPoint(`
	`  Mouse.CursorPos).MonitorNum`

TMouse

Компонент Mouse позволяет получить информацию о текущем состоянии мышки. Основные свойства TMouse представлены в таблице 20.

Таблица 20 – Основные свойства TMouse

Свойства	Описание
Mouse.**CursorPos**: TPoint; Mouse.**CursorPos**.**X**: Integer; Mouse.**CursorPos**.**Y**: Integer;	Положение курсора мышки на экране
Mouse.**MousePresent**: Boolean; Mouse.**WheelPresent**: Boolean;	Сообщают, установлена ли в системе мышка и есть ли у нее колесико
Mouse.**WheelScrollLines**: Integer;	Указывает количество строк, прокручиваемое за каждый поворот колесика мышки

5. РАБОТА С ГРАФИКОЙ

5.1. Методы для работы с графикой

Основные методы для рисования графических объектов (фигур) представлены в таблице 21.

Таблица 21 – Основные методы для рисования графических объектов

Процедура	Описание
procedure **Rectangle**(X1, Y1, X2, Y2: Integer);	Рисует прямоугольник или квадрат (в случае, когда стороны равны). Здесь стороны всегда параллельны осям координат. Чтобы нарисовать повернутый прямоугольник, необходимо использовать Polyline.
procedure **Ellipse**(X1, Y1, X2, Y2: Integer);	Рисует эллипс или круг (в случае, когда стороны равны). Фигура рисуется как вписанная в прямоугольник, который мог бы быть построен по тем же координатам X1, Y1, X2, Y2.
procedure **RoundRect**(X1, Y1, X2, Y2, X3, Y3: Integer);	Рисует прямоугольник со скругленными углами. Радиусы скругления задаются параметрами X3 и Y3. *В остальном совпадает с процедурой* ***Rectangle***.
procedure **TextOut**(X, Y: Integer; Text: String);	Рисует текстовую надпись. Текст располагается ниже и правее указанной точки. Здесь текст может быть только однострочным.
procedure **MoveTo**(X, Y: Integer); procedure **LineTo**(X, Y: Integer);	Рисует линию. Подробнее см. далее в разделе **5.4**.

5.2. Основы работы с графикой

Рисование фигур осуществляется на «Холсте» (**Canvas**). Например, для вывода эллипса необходимо написать:

```
Image1.Picture.Bitmap.Canvas.Ellipse(0, 0, 100, 100);
```

Этот длинный «маршрут» имеет следующие составляющие:

- **Image** – «Изображение» (точнее, «Компонент для отображения изображения»);
- **Picture** – «Картинка», которая будет отображена на Компоненте;
- **Bitmap** – «Битовая карта», «Битовый массив», «Битовая матрица» – внутренняя структура растровой картинки в виде двумерного массива (матрицы) точек, называемых пикселями. Именно здесь

хранится само изображение. *На самом деле, каждый пиксель не обязан состоять только из одного бита, обычно он состоит из 24 бит (16 млн. цветов);*

!!! Имя формата изображения **BMP** и расширение файла ***.bmp** являются сокращением от **Bitmap**.

– **Canvas** – «Канва», «Холст» на котором рисуется (растровая) картинка для отображения на Компоненте.

Размеры изображения

Если добавить на форму компонент **Image** и сразу попытаться рисовать на нем перечисленные выше фигуры, то ничего не отобразится.

Проблема в том, что начальные размеры изображения **0x0**. Вначале необходимо задать размеры битовой матрицы, например:

```
Image1.Picture.Bitmap.Width := 300;
Image1.Picture.Bitmap.Height := 200;
```

Пример

Рассмотрим пример рисования графических объектов. Для этого добавим на форму компонент TImage и кнопку, для которой напишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //Задаем размеры изображения
  Image1.Picture.Bitmap.Width := 300;
  Image1.Picture.Bitmap.Height := 200;
  //Рисуем фигуры
  Image1.Picture.Bitmap.Canvas.Ellipse(0, 0, 100, 100);
  Image1.Picture.Bitmap.Canvas.Rectangle(50, 50, 150, 150);
  Image1.Picture.Bitmap.Canvas.TextOut(55, 55, 'Привет!');
end;
```

После нажатия кнопки получаем изображение, представленное на рисунке 5.1.

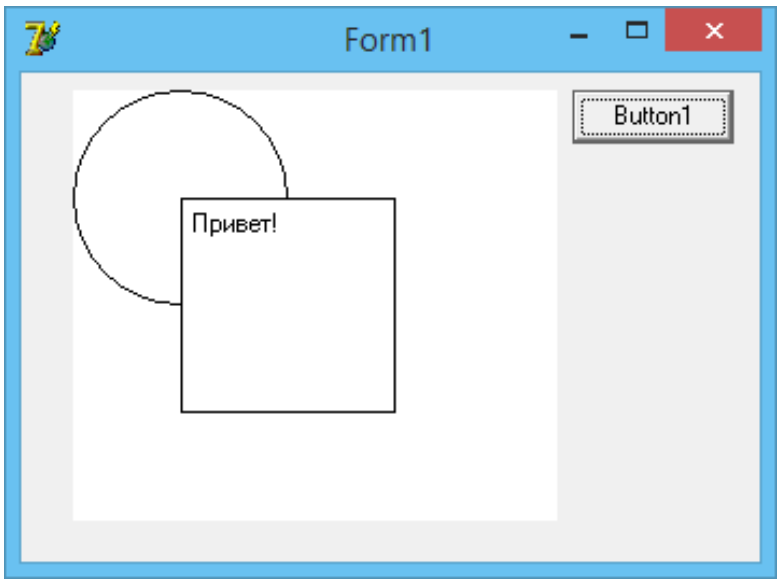


Рисунок 5.1 – Пример рисования

Результат может зависеть от порядка выполнения команд. Так если поменять местами **Ellipse** и **Rectangle**, то круг будет нарисован поверх квадрата (рис. 5.2).

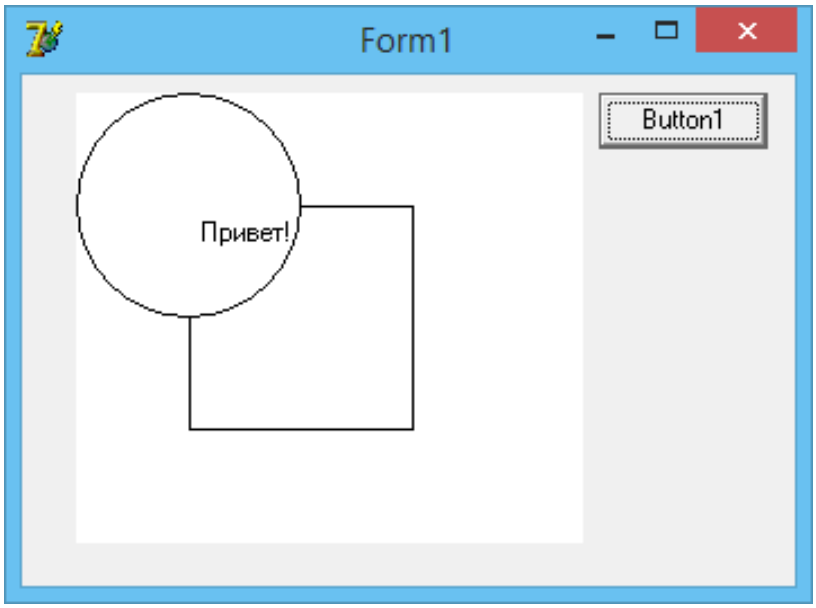


Рисунок 5.2 – Изменение порядка рисования

5.3. Цвета и стили рисования

Для Канвы (Холста) можно задавать цвета для трех различных объектов:

- цвет линий (цвет контура фигур);
- цвет заливки фигур;
- цвет текста.

Кроме того, у каждого из объектов имеются другие настройки стилей рисования.

Перо Pen

Pen (Перо) отвечает за стиль и цвет отображения линии или контура вокруг фигуры. Например, задать цвет линии можно следующей командой:

```
Image1.Picture.Bitmap.Canvas.Pen.Color := clRed;
```

!!! Эту команду нужно выполнить **до** рисования фигуры**!**

Свойство пера **Style** может принимать значения, представленные в таблице 22.

Таблица 22 – Значения Style для Pen

Значение	Описание
psSolid ————	Сплошная линия
psDash — — —	Штриховая линия («тире»)
psDot ----------	Пунктирная линия («точка»)
psDashDot — · — · — ·	Штрихпунктирная линия («тире-точка»)
psDashDotDot — · · — · ·	«Тире и две точки»
psClear	Прозрачная (отсутствие линии)

Например, задать тип линии можно следующей командой:

```
Image1.Picture.Bitmap.Canvas.Pen.Style := psSolid;
```

Также можно задать толщину линии, например:

```
Image1.Picture.Bitmap.Canvas.Pen.Width := 2;
```

!!! Стоит обратить внимание, что задать толщину можно только для сплошной линии**!** Линии, состоящие из «точек» и «тире», могут иметь только толщину «1». *Толщина линии имеет приоритет, если задать одновременно, например, Pen.Width := 2 и Pen.Style := psDash, то линия будет сплошной.*

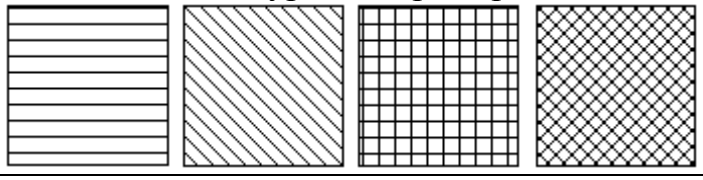
Кисть Brush

Brush (Кисть) позволяет задать цвет и стиль заливки фигуры. Например, задать цвет заливки можно следующей командой:

```
Image1.Picture.Bitmap.Canvas.Brush.Color := clSilver;
```

У кисти есть два основных стиля (табл. 23) и несколько дополнительных текстур *(которыми мы не будем пользоваться)*.

Таблица 23 – Значения Style для Brush

Значение	Описание
bsSolid	Заливка сплошным цветом
bsClear	Без заливки (для фигур рисуется только рамка)
bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross	Различные текстуры. Например: 

Шрифт Font

Задать цвет текста (цвет шрифта) можно следующей командой:

```
Image1.Picture.Bitmap.Canvas.Font.Color := clRed;
```

Аналогичным образом можно задать размер шрифта Font.Size и стиль шрифта Font.Style (жирный, курсив, подчеркнутый, зачеркнутый).

5.4. Рисование линий

Для рисования линий используется команда LineTo, например:

```
Image1.Picture.Bitmap.Canvas.LineTo(10, 20);
```

Эта команда проводит линию **в** точку с координатами X = 10 и Y = 20. Та точка, **из** которой проводится линия, здесь не указывается. У канвы есть «невидимый» курсор (**PenPos**), который указывает, **из** какой точки проводится линия. После выполнения команды LineTo, этот курсор сдвигается в новую точку – в ту точку, **в** которую была проведена линия. *Когда на канве еще не рисовалась ни одна линия, курсор имеет координаты X = 0 и Y = 0.*

Чтобы сдвинуть курсор без рисования линии, используется команда MoveTo.

Для рисования отдельной линии необходимо последовательно выполнить две команды:

```
Image1.Picture.Bitmap.Canvas.MoveTo(X1, Y1);
Image1.Picture.Bitmap.Canvas.LineTo(X2, Y2);
```

В данном случае, линия будет проведена из точки 1 (X1, Y1) в точку 2 (X2, Y2).

* Дополнительные пояснения

Может показаться странным, что нет единой команды для рисования линии, чего-то вроде:

```
...Canvas.Line(X1, Y1, X2, Y2);
```

Но LineTo предназначена в основном не для рисования отдельных линий, а для рисования сложных фигур.

Например, чтобы нарисовать фигуру с четырьмя сторонами (при помощи нашей **выдуманной** команды Line), нам бы пришлось написать:

```
...Canvas.Line(X1, Y1, X2, Y2);
...Canvas.Line(X2, Y2, X3, Y3);
...Canvas.Line(X3, Y3, X4, Y4);
...Canvas.Line(X4, Y4, X1, Y1);
```

Здесь каждая координата использована дважды.

Но для LineTo мы пишем:

```
...Canvas.MoveTo(X1, Y1);
...Canvas.LineTo(X2, Y2);
...Canvas.LineTo(X3, Y3);
...Canvas.LineTo(X4, Y4);
...Canvas.LineTo(X1, Y1);
```

Хотя в данном случае и получается на одну строчку кода больше, но все координаты, кроме первой, используются только один раз. *Да и первая координата используется дважды только в случаях, когда фигура замкнутая.*

!!! С помощью LineTo можно рисовать различные фигуры, в том числе квадраты, прямоугольники и ромбы, но этот способ не позволяет их закрашивать.

5.5. Задачи

Задача 1

Нарисовать поле для игры в крестики-нолики. Для начала простейший вариант **3x3**. Размеры клеток задаются параметром **d** (рис. 5.3), который по умолчанию равен **20** пикселям.

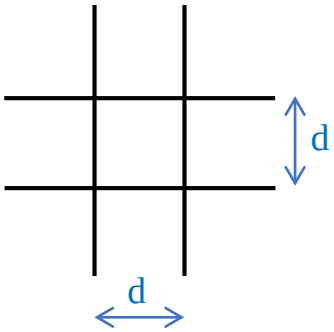


Рисунок 5.3 – Поле для игры

Задача 2

Усложнить предыдущую задачу, нарисовав поле размером **NxM**. По умолчанию считаем **N** = **6**, **M** = **5**, но алгоритм должен быть универсальным и работать при любых **N** и **M**.

Задача 3

Нарисовать шахматную доску (по умолчанию **8x8**).

5.6. РАБОТА № 3
«Разработка графического редактора»

Разработать простейший графический редактор, позволяющий рисовать разными цветами базовые геометрические фигуры (линии, круги, прямоугольники и т.п.), а также выводить текстовые надписи.

Минимальные требования к программе:

- отображение текущих координат курсора мышки;
- кнопки выбора рисуемых фигур (линия, эллипс, прямоугольник, прямоугольник со скругленными углами, текст). Только одна из кнопок может быть зажата;
- выбор цвета рисования линий и текста;
- поле или окно для ввода рисуемого текста;
- иконки кнопок и меню, всплывающие подсказки (на русском!);

– окно «О программе», содержащее информацию о программе, авторе, годе разработки, а также «логотип программы», загруженный в компонент TImage;
– пункт меню и кнопка создания нового документа (очистка изображения).

Начало работы

Добавить на форму следующие компоненты (со вкладок «**Standard**», «**Additional**» и «**Win32**»):

– **Image** (задать ему свойство Align = **alClient** и изменить тип курсора на **crCross**);
– **MainMenu** (добавить в него раздел «Файл» и «Справка»);
– строку состояния **StatusBar** (добавить в нее 3 панели);
– ImageList;
– панель **ToolBar** (связать ее с **ImageList** и добавить на нее 5 кнопок).
– на панель ToolBar добавить выпадающий список с цветами **ColorBox** (в его свойстве **Style** отключить системные цвета и включить **CustomColor**).

В итоге мы получим внешний вид, аналогичный представленному на рисунке 5.4.

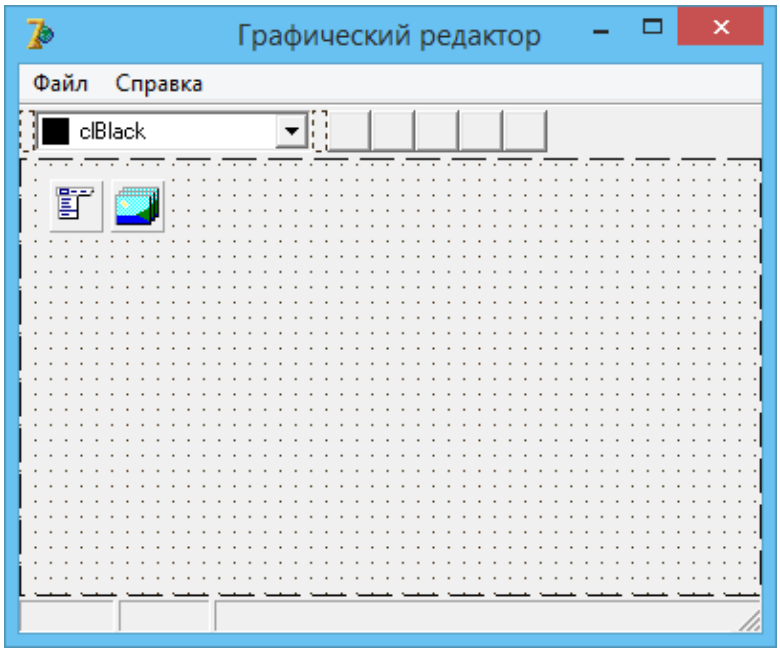


Рисунок 5.4 – Внешний вид программы

Далее необходимо написать следующий код:

– в событии **OnCreate** формы задать размеры изображения:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  //Задаем размеры изображения
  Image1.Picture.Bitmap.Width := 300;
  Image1.Picture.Bitmap.Height := 200;
end;
```

– при наступлении события **OnMouseMove** для Image1 отображать текущие координаты курсора мышки в строке состояния:

```
procedure TForm1.Image1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  //Отобразить текущие координаты
  StatusBar1.Panels[0].Text := IntToStr(X);
  StatusBar1.Panels[1].Text := IntToStr(Y);
end;
```

– при наступлении события **OnMouseDown** для Image1 запоминать координаты нажатия мышки:

```
var X0: Integer = -1;
    Y0: Integer = -1;
procedure TForm1.Image1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Запоминаем координаты нажатия мышки
  X0 := X;
  Y0 := Y;
end;
```

Здесь использованы глобальные переменные **X0** и **Y0**, которые необходимо объявить **до** соответствующей процедуры. Этим переменным присваивается начальное значение -1.

!!! Значение -1 будет использоваться для того, чтобы показать, что сейчас рисование не производится, т.е. кнопка мышки не зажата.

– при наступлении события **OnMouseUp** для Image1 рисуем эллипс по указанным мышкой координатам:

```
procedure TForm1.Image1MouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Устанавливаем цвета для рисования
  Image1.Picture.Bitmap.Canvas.Pen.Color := ColorBox1.Selected;

  //Рисуем Эллипс (или Круг)
  Image1.Picture.Bitmap.Canvas.Ellipse(X0, Y0, X, Y);
end;
```

Цвет линии выбирается из выпадающего списка при помощи компонента **ColorBox**.

Рассмотренный вариант имеет недостаток, заключающийся в том, что фигура отображается только в момент отпускания кнопки мышки. Его необходимо исправить так, чтобы фигуры перерисовывались при каждом движении мышки.

5.7. *РАБОТА № 3 (дополнения)

«Разработка графического редактора»

Дополнить графический редактор следующими функциями:

– изменение цвета заливки фигуры и отключение заливки;
– возможность изменения толщины линии;
– изменение размера холста (по умолчанию 300x200);

- отмена последнего действия;
- пункт меню и кнопка сохранения изображения в файл;
- пункт меню и кнопка загрузки изображения из файла;
- копирование изображения в буфер обмена;
- печать изображения.

5.8. Создание копии изображения

Для создания копии изображения в памяти компьютера необходимо создать копию **Bitmap** с изображением. В свою очередь для этого необходимо объявить **глобальную** переменную типа **TBitmap**:

```
//Запоминание предыдущего изображения
var Bmp0: TBitmap;
```

!!! На самом деле, эта переменная может быть и локальной. Просто здесь рассматривается вариант, необходимый нам для разработки графического редактора. Чтобы узнать, в каких случаях нужна глобальная переменная, а в каких достаточно локальной, см. раздел **6.17**.

Тип **TBitmap** является **Классом** (в отличие от обычных типов переменных, таких как Integer, Real, Boolean или String), поэтому нам вначале обязательно необходимо **создать** Экземпляр этого класса, командой **Create**:

```
//Создаем Bitmap для запоминание предыдущего изображения
Bmp0 := TBitmap.Create;
```

По умолчанию это делается в **OnCreate** формы.

Если этого не сделать, то при попытке выполнить следующие команды, мы получим сообщение об ошибке (рис. 5.5).

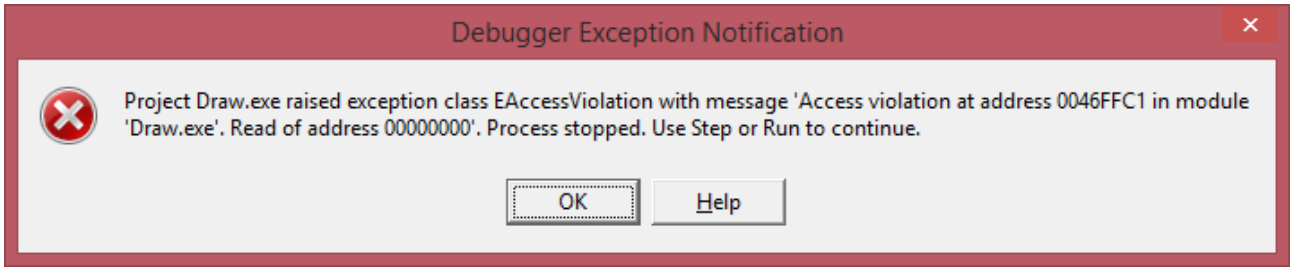


Рисунок 5.5 – Сообщение об ошибке

!!! Подробнее про Классы и Экземпляры классов см. Главу «6. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ».

Для того чтобы запомнить изображение **из Image** → в память, необходимо выполнить следующую команду:

```
//Запоминаем изображение
Bmp0.Assign(Image1.Picture.Bitmap);
```

Обратное действие, восстановление изображения **в Image** ← из памяти, выполняется следующей командой:

```
//Восстанавливаем запомненное изображение
Image1.Picture.Bitmap.Assign(Bmp0);
```

5.9. Очистка холста

У Канвы (Холста) нет свойства Clear для очистки, но очистить изображение все же можно, для этого есть несколько способов:

1. Когда Canvas вложен в Bitmap, то достаточно установить один (любой) из размеров Bitmap в ноль, и сразу вернуть его в предыдущее значение, например:

```
Image1.Picture.Bitmap.Height := 0;
Image1.Picture.Bitmap.Height := 200;
```

2. Либо можно присвоить пустой Bitmap:

```
Image1.Picture.Bitmap.Assign(nil);
```

3. Кроме того, можно рисовать белый прямоугольник поверх всего содержимого холста:

```
...Canvas.FillRect(...Canvas.ClipRect);
```

где FillRect – окрасить заданную область в цвет кисти;
ClipRect – текущий размер всего холста.

!!! Последний способ самый универсальный, т.к. работает непосредственно с Canvas (т.е. как при наличии вышестоящего Bitmap, так и без него).

5.10. *Сохранение в файл/Загрузка из файла

Для сохранения и загрузки изображения используются следующие процедуры:

Image1.Picture.**SaveToFile**(FileName);
Image1.Picture.**LoadFromFile**(FileName);

!!! Имя файла **FileName** лучше всего получать из соответствующих диалогов сохранения файла (**TSaveDialog** или **TSavePictureDialog**) и открытия файла (**TOpenDialog** или **TOpenPictureDialog**).

5.11. *Двойная буферизация

При постоянной перерисовке фигур может наблюдаться мерцание изображения на экране. *Особенно это заметно на старых медленных ПК.*

Для решения данной проблемы необходимо включить двойную буферизацию формы. Для этого достаточно в обработчике события **OnCreate** формы написать:

DoubleBuffered := True;

!!! В новых версиях Delphi, это свойство формы вынесено в Инспектор объектов (рис. 5.6), т.е. нужно просто установить соответствующую галочку.

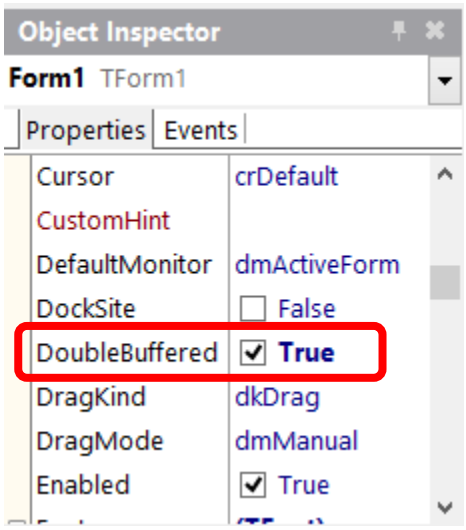


Рисунок 5.6 – Двойная буферизация

Принцип двойной буферизации заключается в наличии двух копий изображения, одна – для чтения (вывода на экран), другая – для записи (рисования фигур). Когда рисование завершено, эти изображения просто меняются местами.

Минусом двойной буферизации является бОльший размер используемой оперативной памяти (незначительный для современных ПК).

6. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Объектно-ориентированное программирование (**ООП** или англ. **ООР**) – парадигма программирования, основанная на концепции объектов. Итоговые программы создаются из объектов, взаимодействующих друг с другом.

Также **Объектно-ориентированное программирование** иногда называют «Методологией программирования». **Методология** – это более широкий термин чем парадигма (парадигма только ее частный случай).

Компоненты (визуальные и невизуальные), такие как кнопки, поля ввода, чекбоксы, радиокнопки и др. *(которые мы рассмотрели ранее)*, являются одним из вариантов Объектов. Т.е. являются наследниками класса TObject. *Далее в качестве Объектов, мы в основном будем рассматривать именно* ***Компоненты****. Но большая часть сказанного будет относиться и к любым другим Объектам.*

ООП предполагает **минимизацию избыточности** данных и защиту их **целостности**.

Основные принципы **ООП**:

- абстракция;
- наследование;
- инкапсуляция;
- полиморфизм.

!!! Последний мы рассматривать не будем, но этот термин нужно знать**!** Об остальных принципах будет рассказано чуть ниже.

К другим терминам, применяемым в ООП, относятся:

- объект;
- класс;
- экземпляр класса;
- предок;
- потомок.

Компоненты (Объекты) состоят из:

- полей данных;
- свойств;
- методов.

В ООП обычно **нет** разделения на Процедуры и Функции. Совместно Процедуры и Функции здесь называются «**Методы**». В Паскале методы объявляются также, как и обычные Процедуры и Функции (начиная со слов «**procedure**» или «**function**»), но используются они **не** как «свободные», а **внутри** Компонента.

Не только процедуры и функции относятся к методам, методами являются:

- процедуры;
- функции;
- конструкторы;
- деструкторы.

Если **Методы** – это процедуры и функции, то **Поля** в ООП – это **переменные**. Они объявляются так же, как и обычные переменные, например:

x: Integer;

но **внутри** Компонента.

Свойства также похожи на **переменные** (имитируют переменные), и для программиста, использующего готовый Компонент, они выглядят как переменные. Но внутри компонента они объявляются немного по-другому. *Хотя с автодополнением кода, которое мы будем применять, граница между Полями и Свойствами становится практически незаметной, и будет видна только при понимании термина «Инкапсуляция».*

Объявление Свойств начинается с ключевого слова «**property**» («свойство»), Методов – со слов «**procedure**» или «**function**», а у Полей **нет** никаких приставок.

На самом деле, со Свойствами и Методами компонентов мы уже встречались ранее, наиболее используемые из них мы выписывали в таблицы.

Некоторые Свойства компонента мы видим через **Инспектор объектов**, но поля и методы в нем никогда не отображаются.

6.1. Создание собственного компонента

Для создания собственного компонента необходимо выполнить следующие действия:

- создать **Пакет** и сохранить его;
- добавить в пакет новый **Компонент**;
- установить пакет;
- **отредактировать** код для компонента;
- перекомпилировать пакет;
- создать **Проект** для тестирования нового компонента;
- **удалить** компонент, когда он больше не нужен.

!!! При работе в компьютерном классе необходимо всегда удалять за собой установленные Компоненты в конце занятия**!** *Это нужно для того, чтобы на этом компьютере могли работать и другие.*

Далее мы будем создавать компонент для игры в **Крестики-нолики**. Это будет визуальный компонент (**Control**).

Данный способ можно применять и для разработки других «плоских» (2D) игр, например, таких как: Шашки, Шахматы, Го, Сапёр, Солитер, а также для создания различных полей для карточных игр (Пасьянс, Дурак, Покер и др.). Конечно, наиболее простыми в данном случае будут Крестики-нолики, Шашки или Го, т.к. игровые фигуры и поле в них можно изобразить из простейших кругов, прямоугольников и линий, а для остальных игр потребуется вначале подготовить библиотеку с соответствующими изображениями.

Создание пакета

Компоненты нужно помещать в отдельный пакет.

!!! Перед созданием пакета, нужно закрыть текущий проект (если он в данное время открыт). Для этого необходимо выбрать меню «File» → «**Close All**» (рис. 6.1).

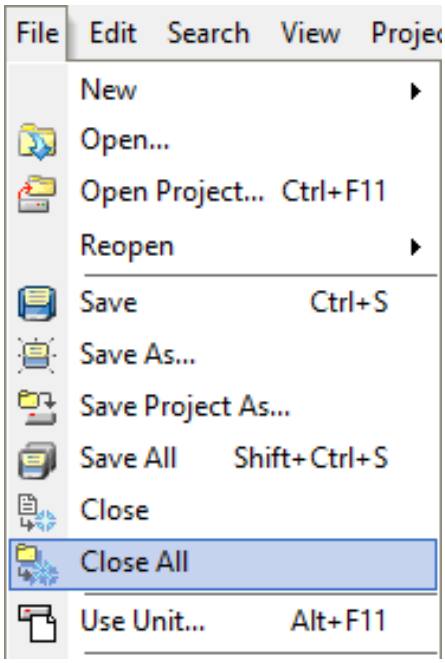


Рисунок 6.1 – Закрытие текущего проекта

Для создания Пакета необходимо выбрать меню «File» → «New» → «**Other**» (рис. 6.2), и выбрать вариант «**Package**» (рис. 6.3).

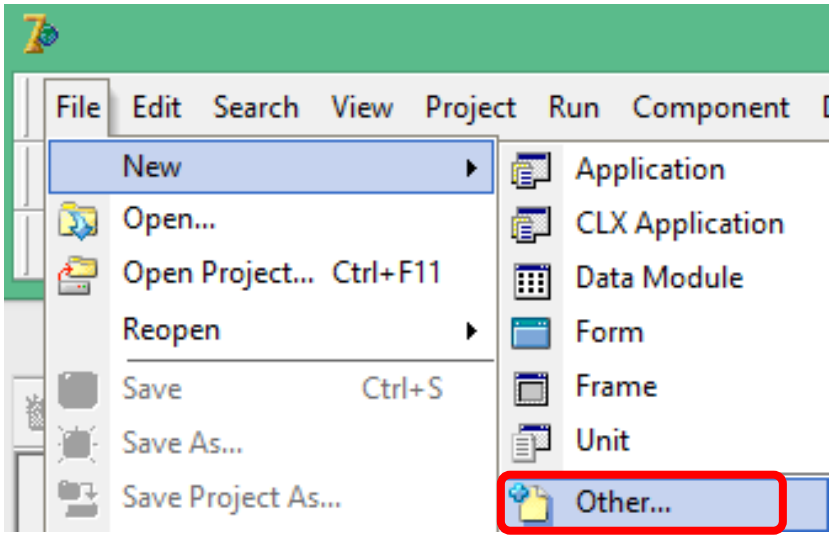


Рисунок 6.2 – Создание другого объекта

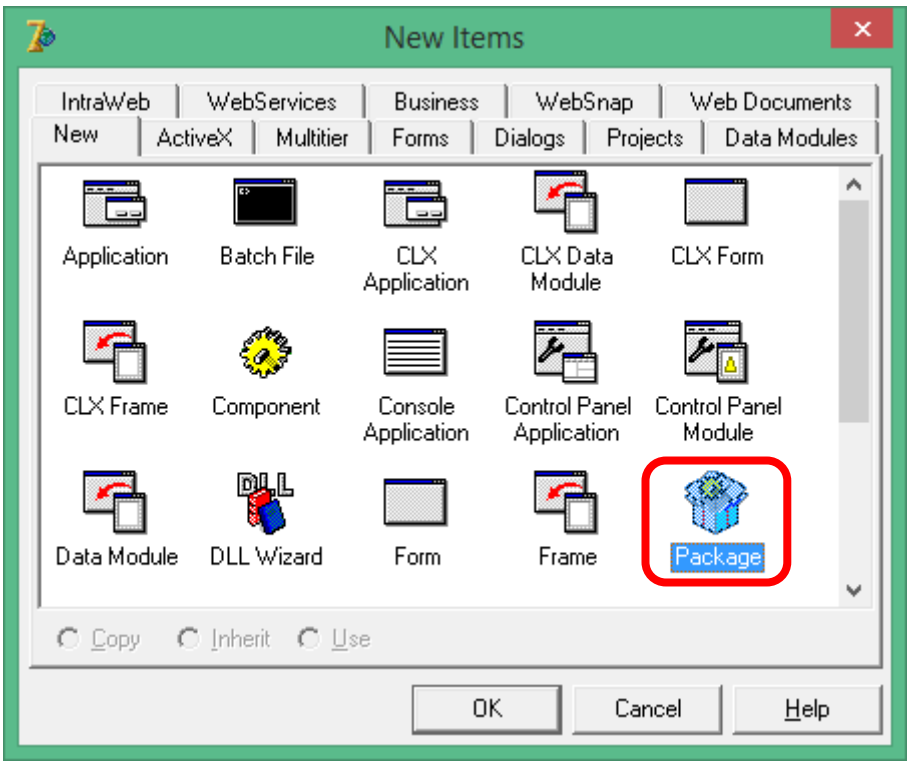


Рисунок 6.3 – Создание Пакета

В результате мы увидим следующее (рис. 6.4).

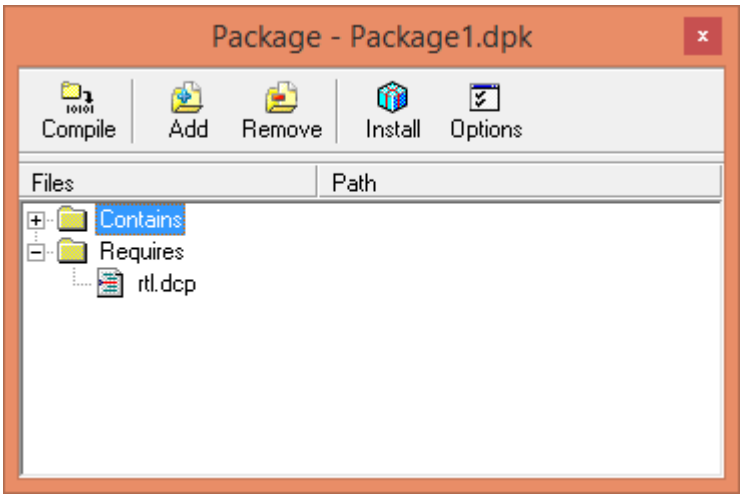


Рисунок 6.4 – Созданный пакет

Теперь необходимо сохранить созданный Пакет в **отдельную папку** (рис. 6.5). В нашем случае это будет папка «**Крестики-нолики**». Пакет назовем «**GameComponents**». *Файл Пакета имеет расширение* ****.dpk***.

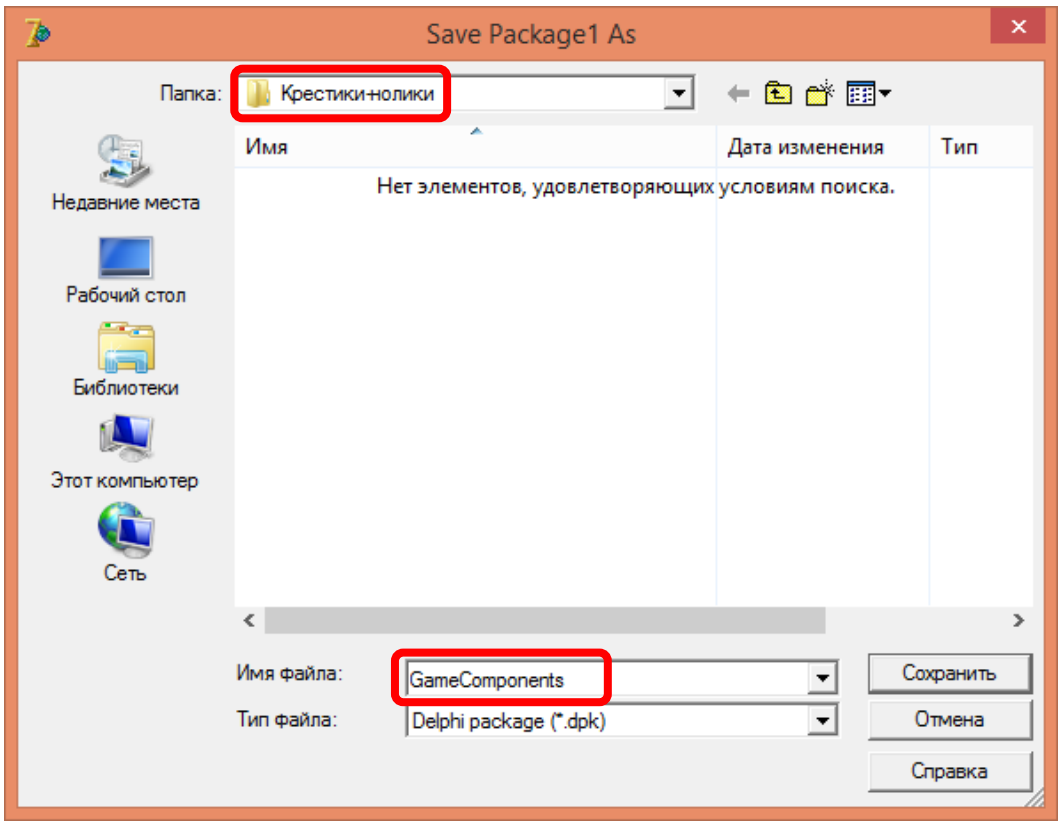


Рисунок 6.5 – Сохранение пакета

Добавление компонента

Для добавления в пакет Компонента необходимо выбрать меню «File» → «New» → «**Other**», и выбрать вариант «**Component**» (рис. 6.6).

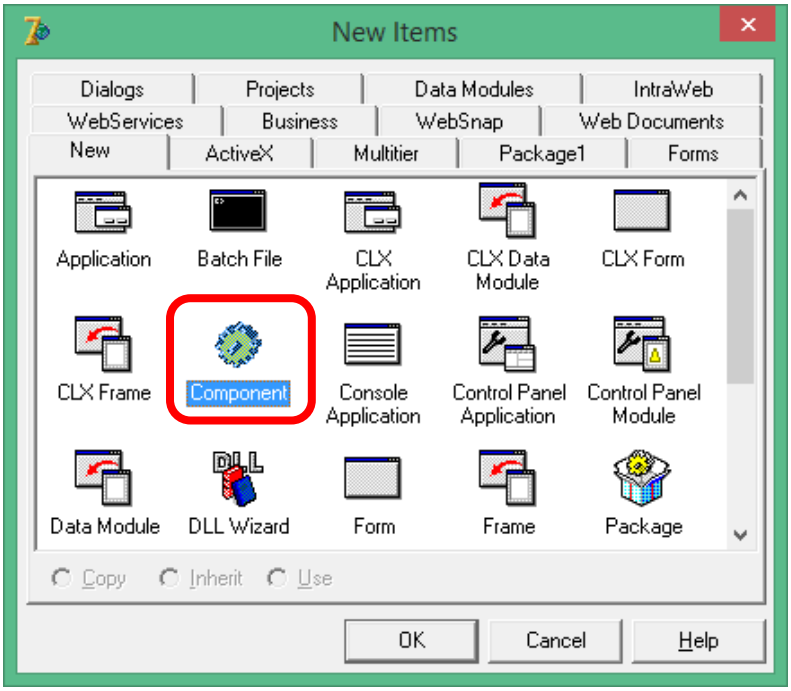


Рисунок 6.6 – Добавление компонента

Далее необходимо указать (рис. 6.7):

- тип **компонента-предка** (в нашем случае это будет **TImage**);
- **класс** (тип) для нового компонента (наш новый компонент назовем «**TControlXO**»). *Стоит напомнить, что типы в **Delphi** принято называть, начиная с приставки «**T**»;*
- выбрать **имя вкладки** в Палитре компонентов, или задать имя для новой вкладки (в нашем случае мы разместим наш компонент на самой первой вкладке «**Standard**»);
- нажать кнопку с многоточием и выбрать туже самую папку, в которую был сохранен Пакет (рис. 6.8). Задать имя модуля «**ControlXO**» (без приставки «**T**»!). *Компоненты сохраняются как отдельные Модули с расширением ***.pas**.*

!!! В один Пакет можно добавить сразу несколько компонентов, причем каждый из них будет в отдельном файле ***.pas**. Хотя здесь рассматривается только создание одного компонента для игры «Крестики-нолики», мы специально назвали Пакет «**GameComponents**» («Игровые компоненты», «Компоненты для игр») во множественном числе, в надежде что в будущем у вас их станет несколько.

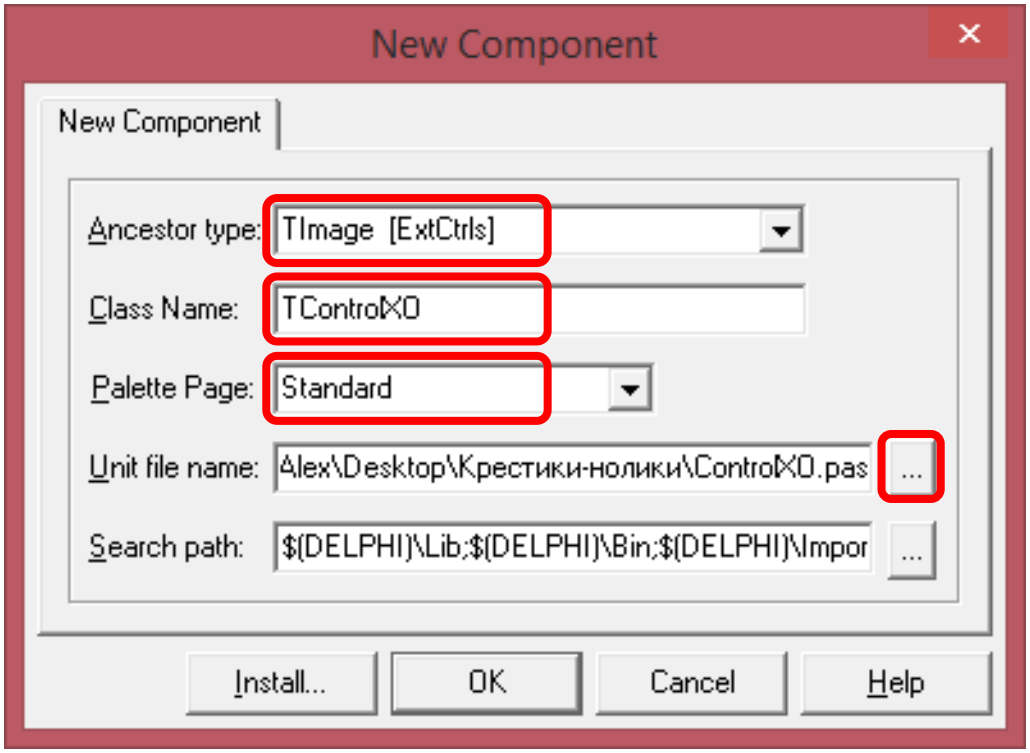


Рисунок 6.7 – Настройка создаваемого компонента

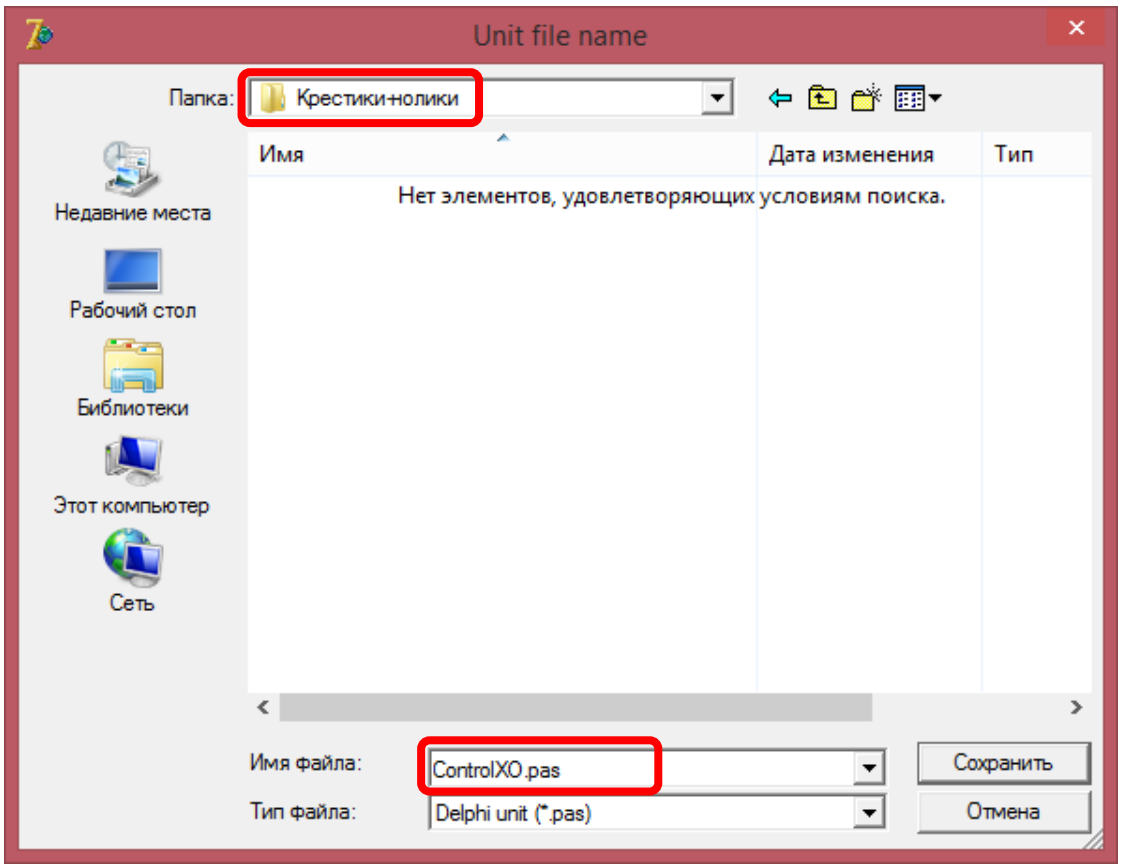


Рисунок 6.8 – Сохранение компонента

Установка пакета

Пакет необходимо установить, нажав кнопку «**Install**» (рис. 6.9).

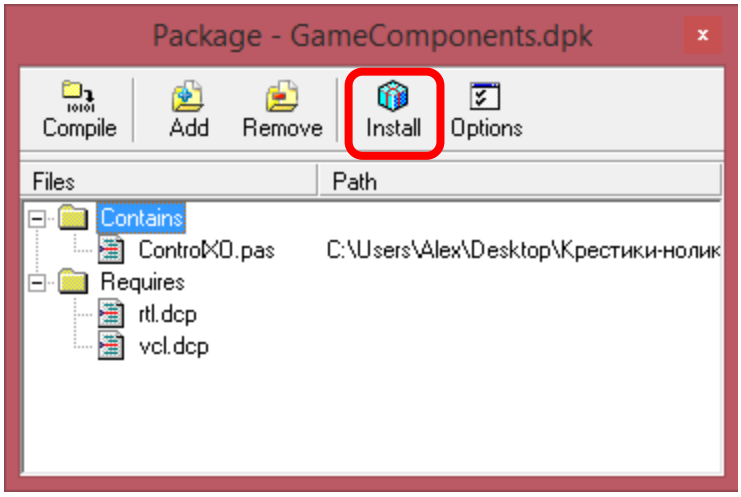


Рисунок 6.9 – Установка Пакета

После Установки в Палитре компонентов появится наш компонент (рис. 6.10). Он имеет то же изображение, что и родительский компонент **TImage**. *Правда, пока данный компонент полностью совпадает с* ***TImage*** *и не делает ничего нового, т.к. мы ничего в него не добавили, поэтому далее требуется написать код для него.*

Рисунок 6.10 – Новый компонент в палитре компонентов

Структура модуля с компонентом

Компонент был добавлен в отдельный Модуль (рис. 6.11). Как и прежде, модуль состоит из двух секций «**interface**» (объявление заголовков) и «**implementation**» («реализация»). В первой из секций уже был добавлен новый тип («**TControlXO**») для нашего Компонента. Все Объекты (в т.ч. Компоненты) объявляются как Классы («**class**»), в скобках указывается тип родительского компонента (в нашем случае это «**TImage**»).

Класс состоит из четырех секций: **private**, **protected**, **public** и **published**. Для начала мы будем использовать только последнюю из них (**published**).

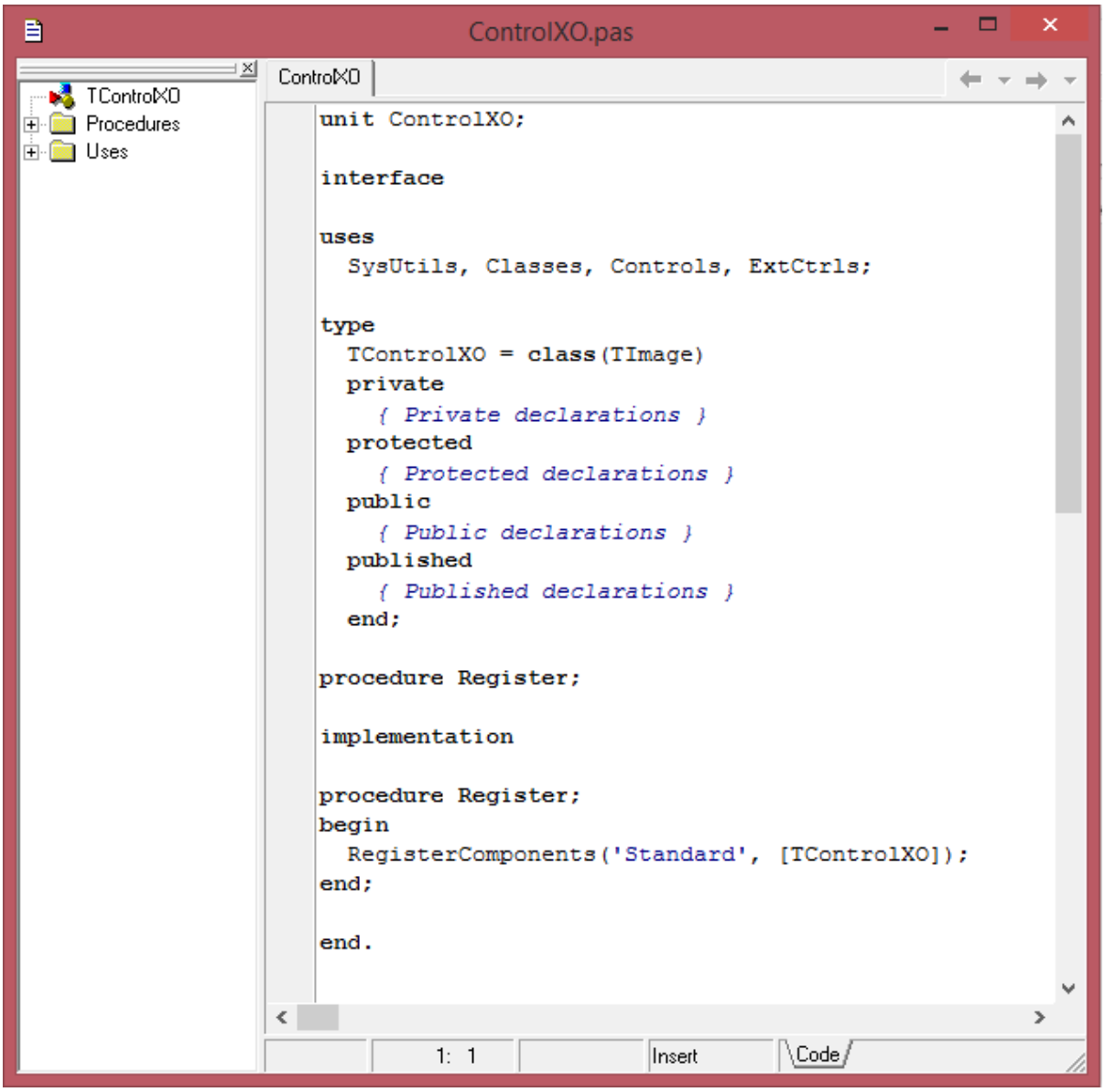


Рисунок 6.11 – Структура модуля с компонентом

Кроме того, в модуль уже добавлена процедура для **регистрации** компонента при установке пакета.

Добавление Свойств к компоненту

Прежде всего, подключим к созданному модулю **дополнительные** библиотеки **Graphics**, **Windows**, **Math**, **Dialogs**. Их нужно дописать в блоке **uses**:

```
uses
  SysUtils, Classes, Controls, ExtCtrls,
  //Модули, подключенные вручную
  Graphics, Windows, Math, Dialogs;
```

Также можно **стереть** комментарии в фигурных скобках, кроме того, можно полностью *удалить* секцию «**protected**», т.к. мы **не** будем ей пользоваться.

Добавим в секцию «**published**» несколько новых Свойств («**property**»). Это будут:

- **CountX** и **CountY** – задающие размеры игрового поля (например, 3x3);
- **CellSize** – размеры каждой ячейки в пикселях (ранее в примере с крестиками-ноликами мы называли его сокращенно как «d»);
- **ColorOfLines** – цвет линий сетки.

В результате мы получим следующий код для Класса:

```
private

public

published
  //Размеры игрового поля
  property CountX: Integer;
  property CountY: Integer;
  //Размер ячейки
  property CellSize: Integer;
  //Цвет линий сетки
  property ColorOfLines: TColor;
end;
```

Далее необходимо нажать «**волшебную**» комбинацию клавиш «**Ctrl+Shift+C**», в результате чего код будет автоматически дополнен:

- к введенным нами Свойствам («**property**») будут добавлены ключевые слова «**read**» и «**write**», а также имена, совпадающие с именем соответствующего свойства, но с приставками «**F**» и «**Set**»;
- для всех этих «**F**» и «**Set**» будут добавлены объявления в секции «**private**». Причем «**F**» объявляются как переменные (Поля), а «**Set**» как процедуры (Методы);

В итоге описание нашего компонента примет следующий вид:

```
private
  FCountX: Integer;
  FCountY: Integer;
  FCellSize: Integer;
  FColorOfLines: TColor;
  procedure SetCellSize(const Value: Integer);
  procedure SetColorOfLines(const Value: TColor);
  procedure SetCountX(const Value: Integer);
  procedure SetCountY(const Value: Integer);
public
published
  //Размеры игрового поля
  property CountX: Integer read FCountX write SetCountX;
  property CountY: Integer read FCountY write SetCountY;
  //Размер ячейки
  property CellSize: Integer read FCellSize write SetCellSize;
  //Цвет линий сетки
  property ColorOfLines: TColor read FColorOfLines
                                write SetColorOfLines;
end;
```

Кроме того, для всех процедур «**Set**», будет добавлена их «**реализация**» в секции «**implementation**»:

```
implementation

...

{ TControlXO }

procedure TControlXO.SetCellSize(const Value: Integer);
begin
  FCellSize := Value;
end;

procedure TControlXO.SetColorOfLines(const Value: TColor);
begin
  FColorOfLines := Value;
end;

procedure TControlXO.SetCountX(const Value: Integer);
begin
  FCountX := Value;
end;

procedure TControlXO.SetCountY(const Value: Integer);
begin
  FCountY := Value;
end;
```

Ввод начальных значений

Для новых Свойств необходимо задать начальные значения (в дальнейшем значения можно будет менять через Инспектор объектов).

В секции «**public**» необходимо объявить Конструктор **Create**:

```
public
  constructor Create(AOwner: TComponent); override;
```

Конструктор – это специальный вид метода *(похож на процедуры и функции)*, который выполняется в момент создания Компонента. В нем можно

задать начальные значения для текущего Компонента (Объекта) и создать другие Объекты, если они требуются для работы данного Компонента.

Теперь снова нажимаем «**волшебную**» комбинацию клавиш «**Ctrl+Shift+C**», в результате чего в секции «**implementation**» будет добавлена «**реализация**» для Конструктора:

```
constructor TControlXO.Create(AOwner: TComponent);
begin
  inherited;

end;
```

Внутри «**begin**» и «**end**» уже имеется обязательная команда «**inherited**» («унаследованный»), все остальные команды нужно писать только **после** нее.

Зададим следующие начальные значения:

- размеры игрового поля – **3x3**;
- размеры каждой ячейки – **50** пикселей;
- цвет линий сетки – **черный**.

В результате мы получим следующий код для Конструктора:

```
constructor TControlXO.Create(AOwner: TComponent);
begin
  inherited;
  FCountX := 3;
  FCountY := 3;
  FCellSize := 50;
  FColorOfLines := clBlack;
end;
```

Кроме конструктора существует еще **Деструктор**:

```
destructor Destroy; override;
```

Он работает противоположно конструктору, т.е. выполняется при завершении работы с компонентом, и позволяет уничтожить объекты, которые были созданы конструктором. Если в Конструкторе осуществляется только настройка начальных значений для таких типов как **Integer**, **Real**, **Boolean**, **String**, **TColor** *(как в нашем примере)*, то Деструктор не нужен вообще.

Внутри Деструктора также используется обязательная команда «**inherited**» («унаследованный»), но здесь ее нужно, наоборот, выполнять самой **последней**:

```
destructor TControlXO.Destroy;
begin
  ...
  inherited;
end;
```

В случае, когда мы добавляем компоненты на Форму из Палитры компонентов, т.е. создаем компоненты через **графический редактор**, нам не нужно будет писать **Create** и **Destroy** в коде Проекта, т.к. создание и уничтожение компонентов в этом случае производится **автоматически**.

Компиляция Пакета

Откомпилируем Пакет. Для этого необходимо нажать кнопку «**Compile**» (рис. 6.12) или нажать комбинацию клавиш «**Ctrl+F9**».

!!! После внесения изменений в код, Пакет всегда нужно будет перекомпилировать заново**!**

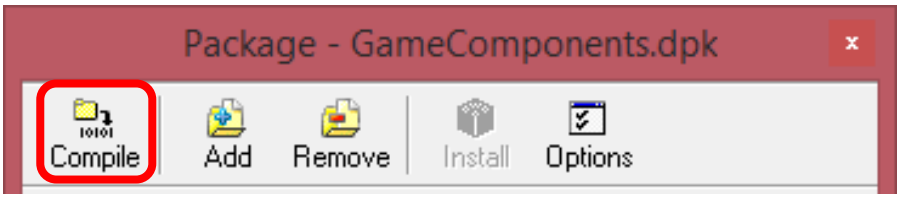


Рисунок 6.12 – Компиляция Пакета

Кроме того, как это было раньше для Проекта, для компиляции можно нажать клавишу «**F9**» или кнопку «**Run**» (рис. 6.13). Единственное, что при таком способе компиляции, появится окно с сообщением (рис. 6.14), которое, впрочем, можно просто закрыть.

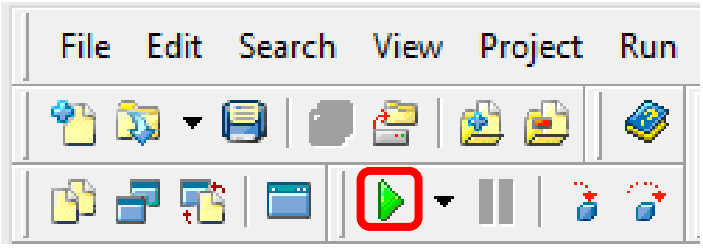


Рисунок 6.13 – Кнопка Run

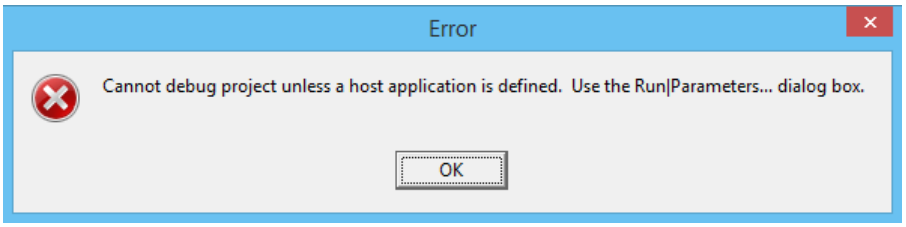


Рисунок 6.14 – Сообщение об ошибке

6.2. Использование и удаление компонента

Добавление созданного компонента в Проект

Теперь, если закрыть Пакет и создать новый Проект с окном, то мы сможем добавить наш новый Компонент из **Палитры компонентов** (см. рис. 6.10). В **Инспекторе объектов** для данного компонента мы увидим добавленные нами свойства **CountX**, **CountY**, **CellSize** и **ColorOfLines** (рис. 6.15), которым будут присвоены указанные нами начальные значения. *При этом все свойства родительского компонента также остались в этом списке.*

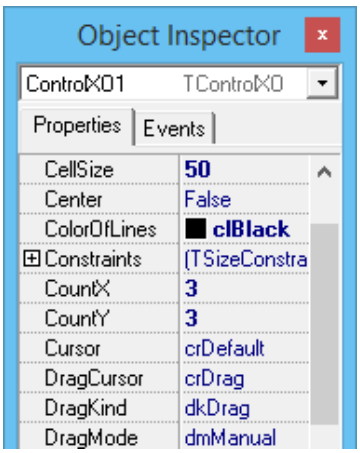


Рисунок 6.15 – Свойства нового компонента

Преимуществами создания собственных Компонентов является то, что:

- написанный для данных Компонентов код больше не загромождает основную программу, т.к. он **вынесен** в отдельный модуль;
- созданные Компоненты можно использовать не только в текущем проекте, но и в любых других проектах (т.е. использовать **повторно**).

К **недостаткам** создания собственных Компонентов относится то, что:

- при переходе на другой компьютер недостаточно перенести только основной проект, необходимо перенести также и пакет с собственными Компонентами и установить данный пакет (без него основной проект не откомпилируется**!**).

!!! В случае, когда в студенческих работах использованы собственные Компоненты, к отчету должен быть прикреплен также исходный код соответствующих Модулей**!**

Написание кода, использующего компонент

Добавим на форму наш новый компонент **ControlXO** и кнопку **Button** (рис. 6.16), после чего сразу **сохраним** проект.

!!! Проект (файлы *.**dpr** и *.**pas**) необходимо сохранить в туже папку, в которую был сохранен компонент**!**

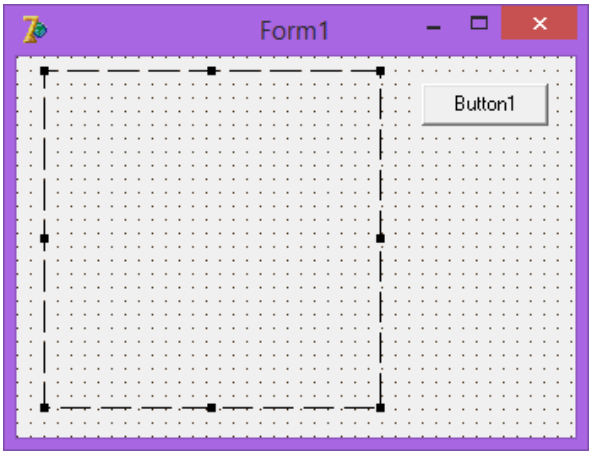


Рисунок 6.16 – Начало создания проекта

Для кнопки напишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var X, Y: Integer;
begin
  //Очищаем старое изображение
  ControlXO1.Picture.Bitmap.Assign(nil);
  //Устанавливаем размер изображения
  ControlXO1.Picture.Bitmap.Width  := ControlXO1.CountX * ControlXO1.CellSize;
  ControlXO1.Picture.Bitmap.Height := ControlXO1.CountY * ControlXO1.CellSize;

  //Цвет линий сетки
  ControlXO1.Picture.Bitmap.Canvas.Pen.Color := ControlXO1.ColorOfLines;
  //Вертикальные линии сетки
  for X := 1 to ControlXO1.CountX - 1 do
  begin
    ControlXO1.Picture.Bitmap.Canvas.MoveTo(X * ControlXO1.CellSize, 0);
    ControlXO1.Picture.Bitmap.Canvas.LineTo(X * ControlXO1.CellSize,
                                            ControlXO1.Picture.Bitmap.Height);
  end;
  //Горизонтальные линии сетки
  ...
end;
```

Если теперь запустить программу (**F9**) и нажать кнопку, то будет нарисована сетка **3x3** (рис. 6.17).

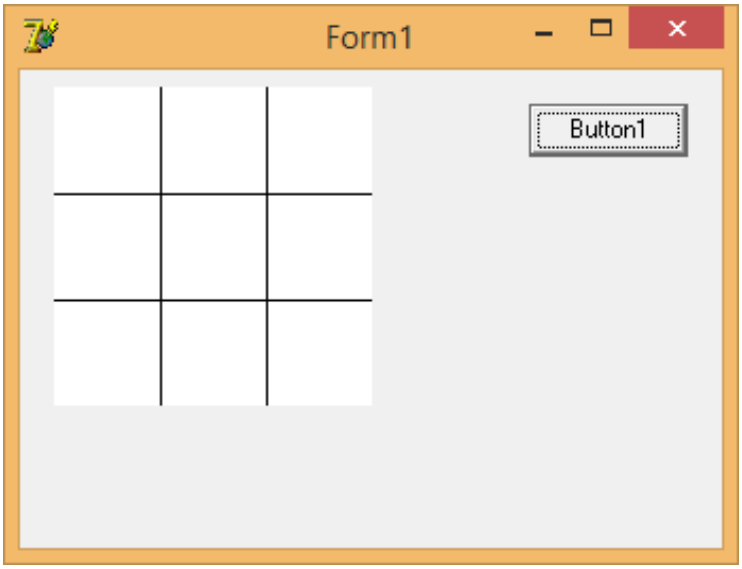


Рисунок 6.17 – Сетка 3x3

Закроем это окно и в Инспекторе объектов (рис. 6.18) изменим настройки сетки на **6x4**, **красного** цвета. Также уменьшим размеры ячеек с 50 до **30**. Если теперь запустить программу (**F9**), то мы увидим новую сетку (рис. 6.19).

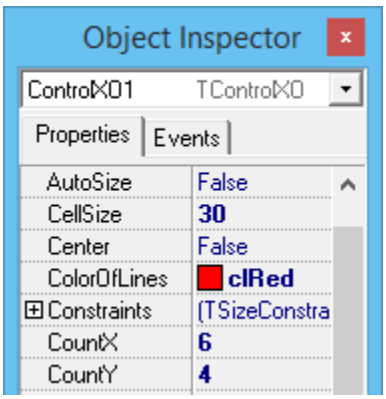


Рисунок 6.18 – Настройка сетки

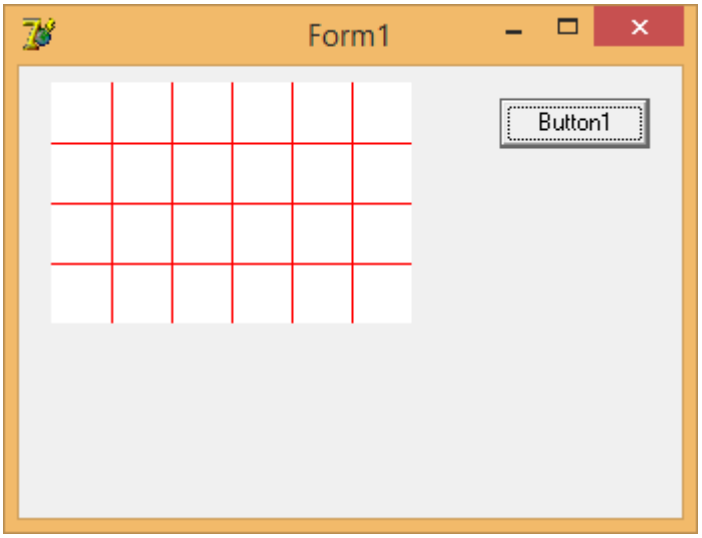


Рисунок 6.19 – Сетка 6x4 красного цвета

!!! При желании можно добавить к компоненту и другие Свойства для настройки его внешнего вида. Например, это может быть **WidthOfLines** – для настройки ширины линий сетки, или **ColorBackground** – для изменения цвета фона.

Удаление пакета

Для удаления Пакета и, соответственно, всех входящих в него Компонентов, необходимо выбрать меню «Component» → «**Install Packages**» (рис. 6.20), далее выбрать из списка требуемый пакет (рис. 6.21) и нажать кнопку удаления («**Remove**»).

!!! Пакеты в списке расположены по алфавиту. Найти добавленный нами пакет очень легко, т.к. мы не задавали для него имя, то его имя будет начинаться с «**c:\delphi7**» (или другого аналогичного имени папки, в зависимости от того, куда изначально был установлен **Delphi**).

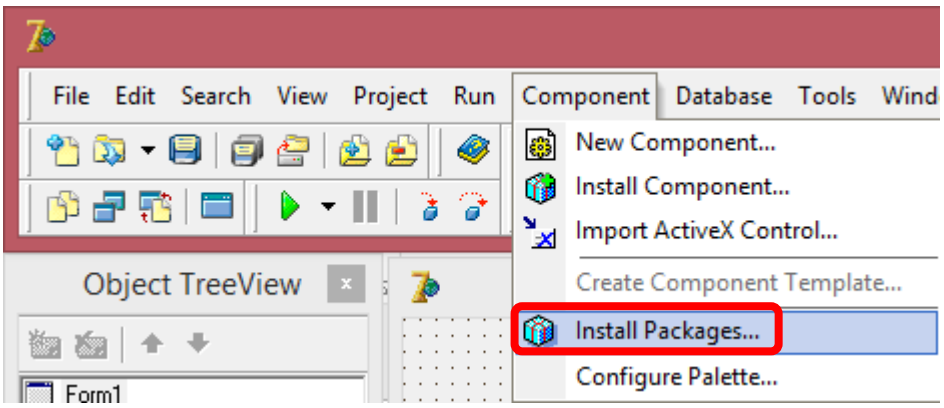


Рисунок 6.20 – Работа с установленными пакетами

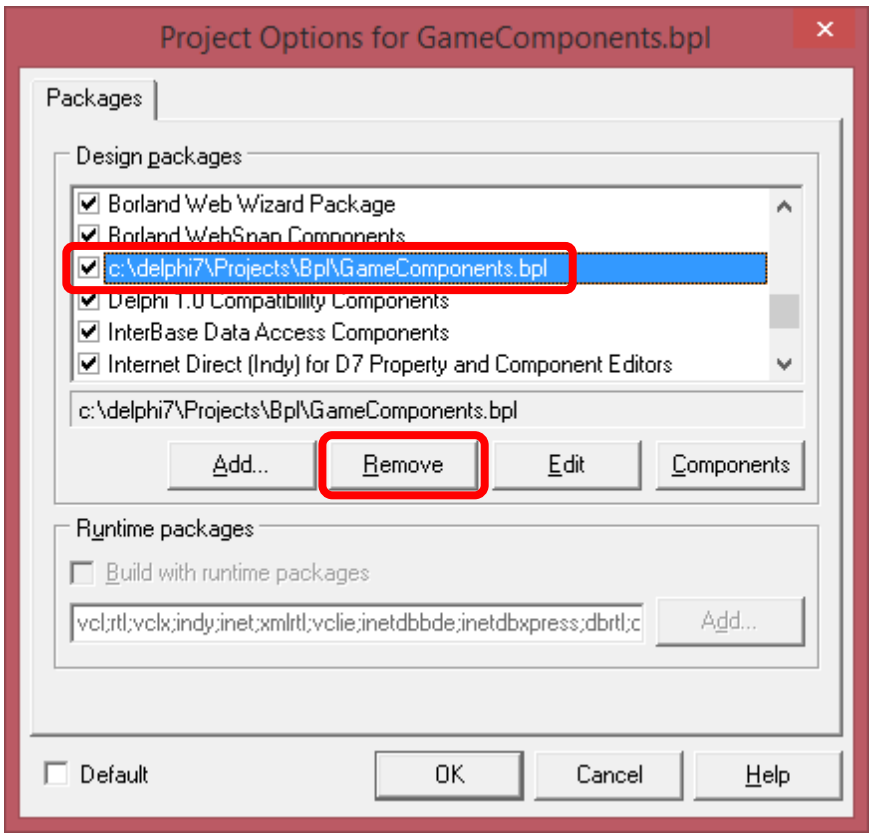


Рисунок 6.21 – Удаление пакета

6.3. *Возможные ошибки

Компонент не установлен

В случае если компонент не установлен, то при открытии Проекта появится сообщение об ошибке (рис. 6.22).

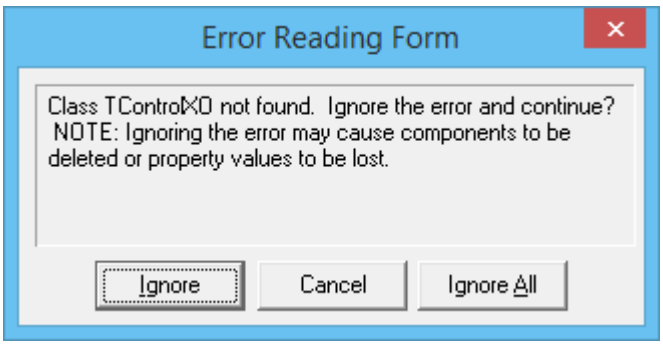


Рисунок 6.22 – Сообщение об отсутствии компонента

В этом случае необходимо закрыть Проект и вначале **установить** компонент.

Аналогичное сообщение может появляться и в случае, когда в Проекте содержатся только стандартные компоненты (нет собственных компонентов), но Проект открывается в более **старой** версии Delphi чем та, в которой он разработан. Это означает, что в старой версии еще не было данных компонентов. В этом случае необходимо использовать более новую версию Delphi.

Уже установлен компонент с тем же именем

Если в системе уже установлен компонент с тем же именем, то мы получим сообщение об ошибке (рис. 6.23) и не сможем установить наш компонент. *Это значит, что на этом компьютере кто-то ранее уже создавал такой же компонент, но не удалил его после себя.*

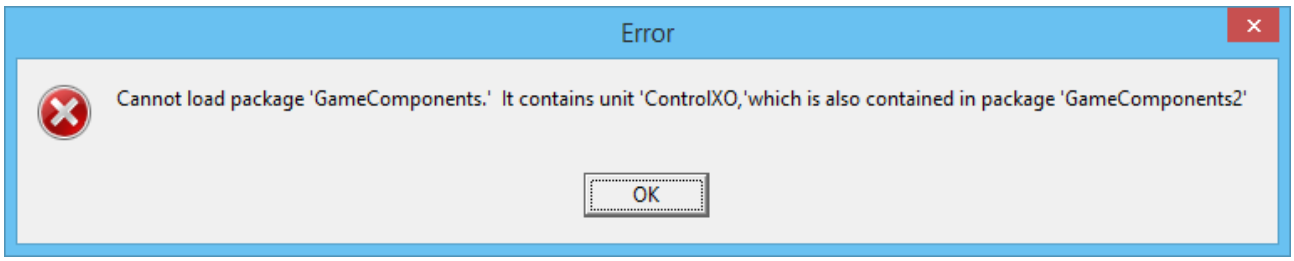


Рисунок 6.23 – Сообщение о наличии компонента

Для решения этой проблемы необходимо удалить мешающий пакет (см. раздел **6.2**).

6.4. *Иконка компонента и имя пакета

Имя пакета

Для того чтобы задать Пакету осмысленное имя, которое будет отображаться в списке установленных пакетов (см. рис. 6.21), необходимо открыть в **Delphi** этот пакет, после чего выбрать меню «Project» → «**Options**» (рис. 6.24), и далее ввести имя пакета в поле «**Description**» (рис. 6.25).

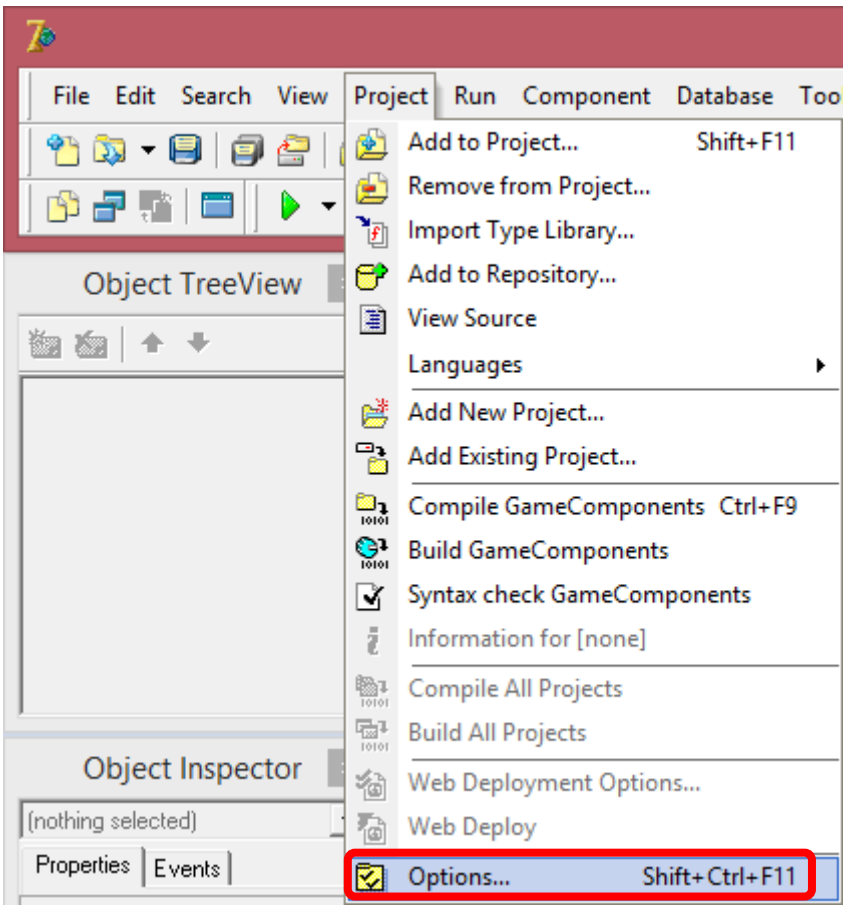


Рисунок 6.24 – Параметры пакета

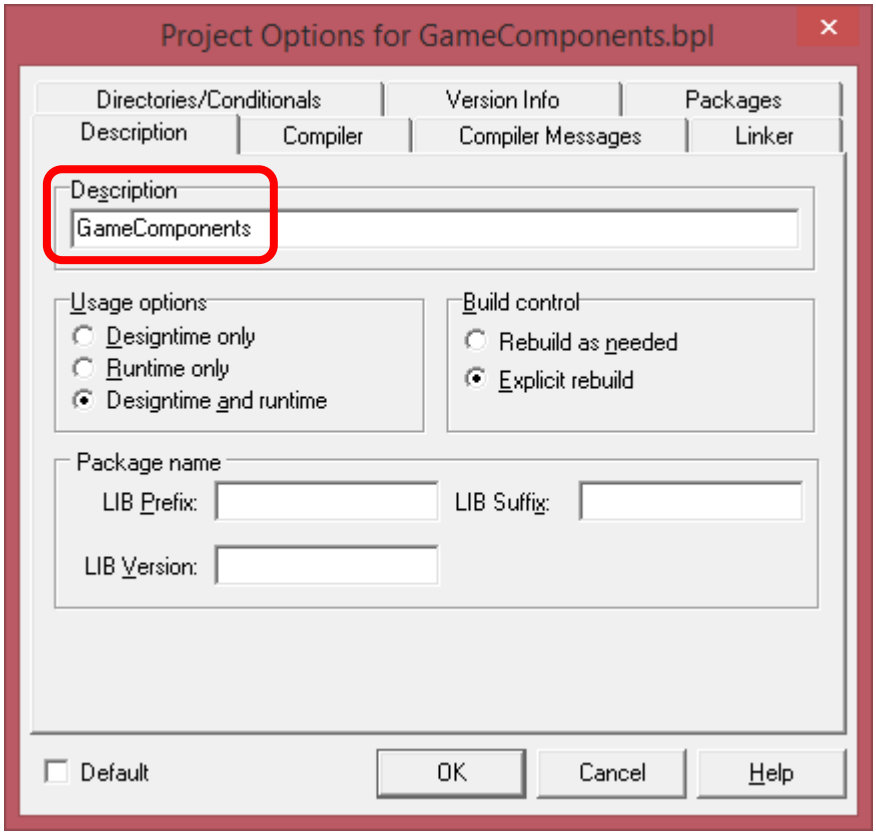


Рисунок 6.25 – Имя пакета

Создание иконки для компонента

Иконка созданного нами компонента выглядит также, как иконка у **TImage** (см. рис. 6.10), от которого мы наследуемся. Но можно добавить собственную иконку для нашего Компонента (рис. 6.26).

Рисунок 6.26 – Собственная иконка

Для этого необходимо:

− в **Delphi 7** выбрать меню Tools → «**Image Editor**».

В новых версиях Delphi нет встроенного редактора и нужно использовать сторонние программы.

− в открывшейся программе создать новый файл «**.dcr**» (рис. 6.27);

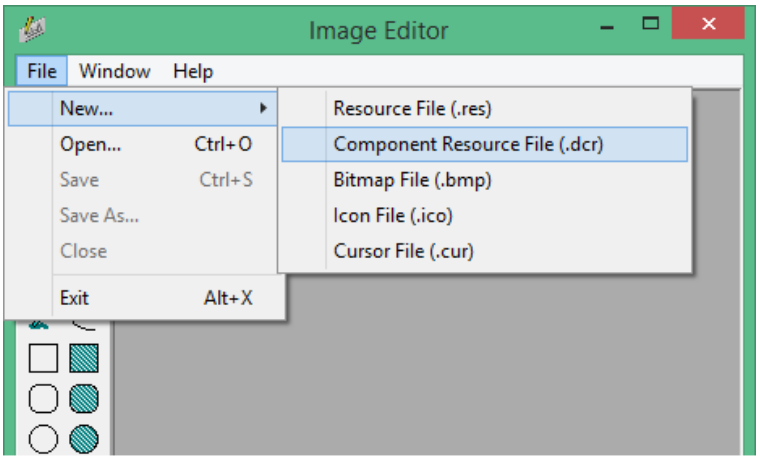


Рисунок 6.27 – Создание файла с ресурсами

− добавить в список новое изображение **Bitmap** (рис. 6.28);

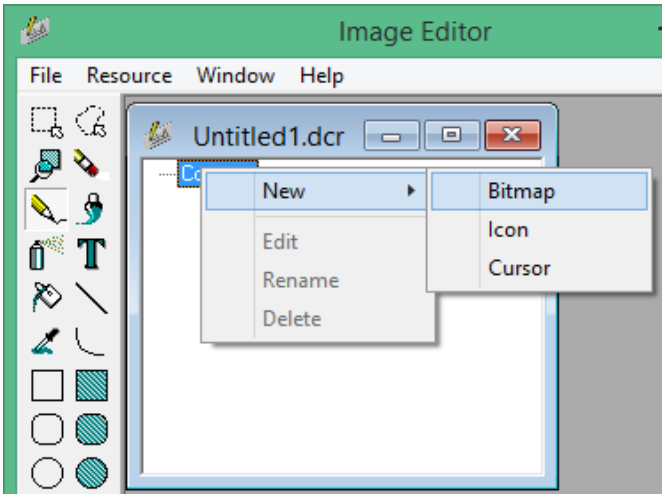


Рисунок 6.28 – Добавление ресурса

– задать размеры изображения **24x24** пикселя (рис. 6.29), **16** цветов;

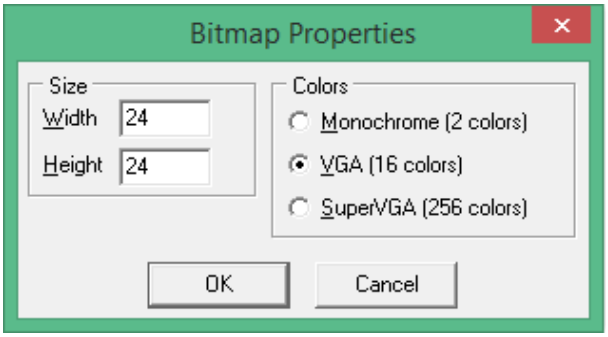


Рисунок 6.29 – Настройки ресурса

– переименовать добавленное изображение (рис. 6.30), задав для него имя, совпадающее с названием Компонента («**TControlXO**»). *Имя изображения будет автоматически исправлено на большие буквы – это нормально.* А также сохранить весь файл, задав для него имя, совпадающее с названием Модуля («**ControlXO.dcr**»). *Файл необходимо сохранить в ту же папку, где расположен соответствующий Модуль (т.е. файл «**ControlXO.pas**»);*

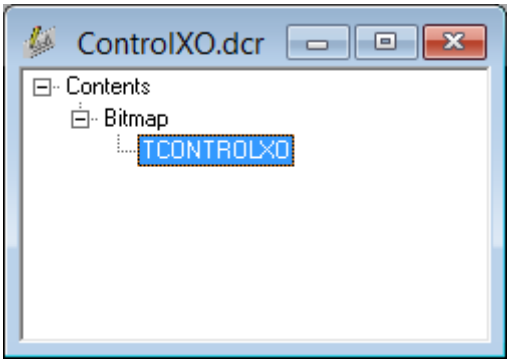


Рисунок 6.30 – Имена изображения и файла

– откроем созданное изображение и отредактируем его (рис. 6.31);

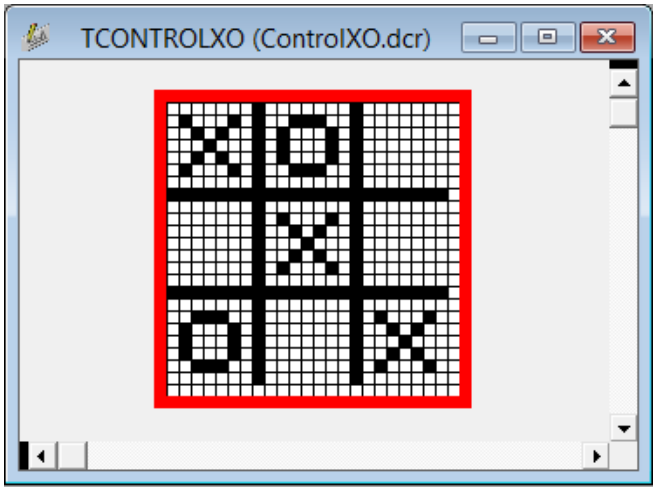


Рисунок 6.31 – Редактирование изображения

– далее необходимо открыть требуемый Пакет (рис. 6.32) и в нем удалить (**Remove**) наш Модуль, и снова добавить (**Add**) этот же Модуль. После повторного добавления файла станет уже два (рис. 6.33), одним из них будет файл с ресурсами (***.dcr**);

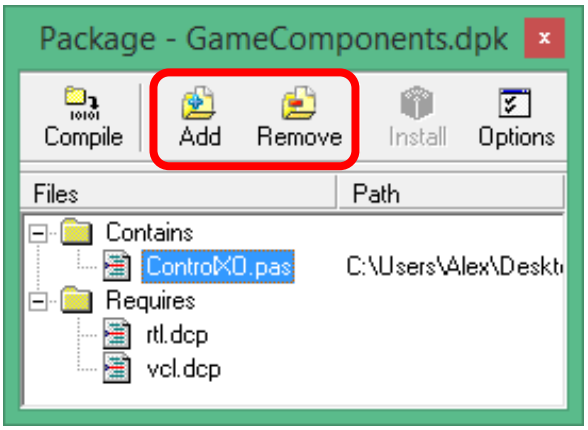


Рисунок 6.32 – Созданный ранее пакет

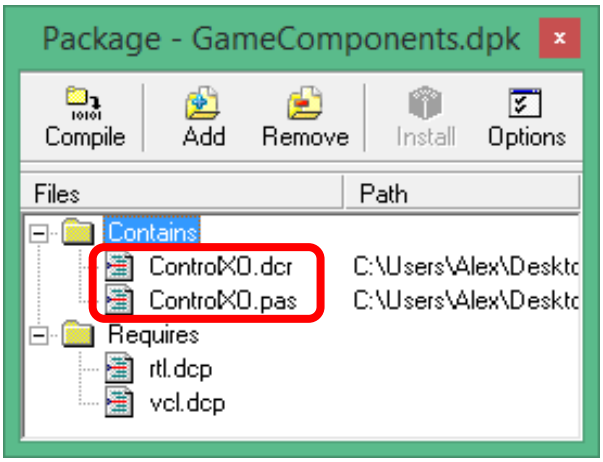


Рисунок 6.33 – Итоговый вид пакета

– перекомпилировать Пакет.

6.5. Наследование

Наследование (англ. **inheritance**) – концепция объектно-ориентированного программирования, согласно которой тип данных может наследовать функциональность некоторого существующего типа.

!!! Наследование может только **расширять** функционал объекта, но не сужать его**!** Например, мы можем добавить к компоненту новые Свойства, но не удалить существующие.

Класс (**class**) – это специальный **Тип** данных, предназначенный для создания Объектов (Компонентов).

Из рассмотренного ранее Классы больше всего похожи на тип Записей (**record**) и являются следующим поколением их развития, также позволяя

работать с разнородными полями, имена которых в коде записываются через точку. Но в отличие от Записей, Классы обладают **Наследованием** и другими дополнительными функциями.

Объект – это то, что сделали «по чертежу», **Класс** – это «чертеж». Хотя, тоже самое можно сказать и про любой другой Тип данных и Переменную этого типа (в этом случае Тип будет чертежом будущей Переменной).

Тот класс, который мы создаем – называется «**наследник**», «**потомок**», «**дочерний класс**», «производный класс» или «подкласс».

Тот класс, на **основании** которого мы создаем новый класс (в нашем примере это «**TImage**») – называется «**предком**», «**родителем**» или «родительским классом». У компонента (объекта) в **Delphi** может быть только **один** предок *(хотя некоторые другие языки допускают множественное наследование)*. При этом количество наследников у компонента не ограничено.

Класс – это, например, **TButton**. **Экземпляры класса** – это **Button1**, **Button2**, **Button3** и т.д. Мы можем и **не** добавить на форму ни одной кнопки, тогда (несмотря на то, что Класс существует) у него имеется **ноль** экземпляров.

Если рассмотреть класс «Кошки», то у всех кошек много общего, например, у них 4 лапы, а не 8 (как для класса «Пауки»), длинные усы и треугольные уши. Но каждая конкретная кошка уникальна и отличается от остальных – это **экземпляр класса**. Более того, существует класс «Динозавры», но экземпляров этого класс в данное время не наблюдается.

Термины **Объект** и **Экземпляр класса** обычно являются синонимами, но слово «Экземпляр» подчеркивает индивидуальность Объекта.

6.6. Перенос кода в компонент

!!! При написании кода для Компонентов всегда нужно быть особенно внимательным, продумывать и осознавать каждый свой шаг. Так, если раньше наш код мог вызвать только ошибку в нашей же программе, то теперь в случае ошибки может «вылететь» вся среда **Delphi**.

Для переноса кода из Проекта в Компонент необходимо выполнить следующие действия:

- **вырезать** из Проекта код для рисования сетки;
- закрыть Проект и открыть Пакет;
- для компонента **TControlXO** в секции **public** прописать новую процедуру **Redraw** («Перерисовать»):

```
procedure Redraw;
```

- нажать «**волшебную**» комбинацию клавиш «**Ctrl+Shift+C**»;
- в появившуюся в секции **implementation** реализацию процедуры **Redraw**, **вставить** скопированный ранее код;
- из вставленного кода **удалить** все «надписи» «**ControlXO1**», т.к. теперь мы находимся на один уровень выше. В итоге мы получим следующий код:

```
procedure TControlXO.Redraw;
var X, Y: Integer;
begin
  //Очищаем старое изображение
  Picture.Bitmap.Assign(nil);
  //Устанавливаем размер изображения
  Picture.Bitmap.Width  := CountX * CellSize;
  Picture.Bitmap.Height := CountY * CellSize;

  //Цвет линий сетки
  Picture.Bitmap.Canvas.Pen.Color := ColorOfLines;
  //Вертикальные линии сетки
  for X := 1 to CountX - 1 do
  begin
    Picture.Bitmap.Canvas.MoveTo(X * CellSize, 0);
    Picture.Bitmap.Canvas.LineTo(X * CellSize, Picture.Bitmap.Height);
  end;
  //Горизонтальные линии сетки
  ...
end;
```

- откомпилировать Пакет;
- закрыть Пакет и вернуться к Проекту;
- в Проекте для события Кнопки (откуда мы ранее вырезали код) написать единственную команду:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ControlXO1.Redraw;
end;
```

- запустить программу (**F9**) и убедиться, что она работает как и прежде, хотя теперь у нас весь код состоит всего из одной строчки.

6.7. Инкапсуляция

Инкапсуляция – механизм языка, ограничивающий доступ одних частей программы к другим. В простейшем случае для **ООП** это сводится к тому, что пользователь (нашего Компонента) **не** должен иметь прямого доступа к переменным Компонента (полям), чтобы не нарушить его работу.

Слово «**инкапсуляция**» происходит от латинского **in capsula** – «размещение в оболочке». Таким образом, инкапсуляцию можно понимать как изоляцию чего-либо инородного с целью исключения влияния на окружающее или выделение основного путем помещения всего мешающего в некую условную капсулу.

Инкапсуляция в **Delphi** (как и во многих других языках) реализуется с применением **Геттеров** и **Сеттеров**, а также при помощи модификаторов доступа (**private**, **protected**, **public**, **published**).

В Delphi существует 4 уровня доступа к методам, полям и свойствам Компонента (Объекта). Они задаются соответствующими Модификаторами доступа:

- **private** («приватный», «частный», «тайный») – то, что объявлено в этой секции, является внутренним делом компонента и **не** будет видно нигде в Коде (кроме методов внутри самого компонента).

- **public** («публичный», «открытый») – видно везде в Коде;

оставшиеся два уровня похожи на предыдущие:

- **published** («опубликовано») – тоже что и **public**, плюс добавляется в Инспектор объектов. В этой секции имеет смысл объявлять только Свойства, но не Поля и Методы, т.к. только Свойства видны в Инспекторе объектов;
- **protected** («защищенный») – средний вариант между **public** и **private**. **Не**видно из Кода проекта, в который добавляется Компонент (как у **private**), но видно наследникам данного компонента (а также из методов внутри самого компонента) *Т.к. мы не собираемся создавать других наследников от нашего Класса, то мы не будем пользоваться этой секцией.*

Секции **public**, **protected** и **private** есть и в некоторых других языках (например, **C++**, **Java**). Модификатор **published** в других языках **отсутствует**, он был добавлен в **Паскаль** при появлении библиотеки **VCL** (именно в связи с необходимостью отображать Свойства в Инспекторе объектов).

Геттеры и **Сеттеры** – это методы, задача которых контролировать доступ к Полям[5]. **Get** – это «получить» (значение). **Set** – это «задать» или «установить» (значение). Геттеры и Сеттеры вместе называются **Аксессоры** (от англ. **Access** – доступ). Геттеры используются после ключевого слова **read** («читать»), а Сеттеры после ключевого слова **write** («писать»).

В **Delphi** принято начинать имя Геттера с приставки **Get**, имя Сеттера с приставки **Set**, а имя Поля (**Field**) с приставки **F**.

Свойство с полным комплектом Аксессоров выглядит следующим образом:

property Abc: Integer read **GetAbc** write **SetAbc**;

Если Геттера **нет**, т.е. чтение производится **напрямую** из Поля:

property Abc: Integer read **FAbc** write **SetAbc**;

Возможна и ситуация с одновременными **прямыми** чтением и записью Поля:

property Abc: Integer read **FAbc** write **FAbc**;

По умолчанию будем считать, что для вновь добавляемого Свойства:

- запись всегда происходит через Сеттер;
- чтение всегда происходит напрямую (**без** Геттера).

Если используется другая комбинация, то автор программы должен доказать ее необходимость.

Кроме того, Свойства могут быть **только для чтения**:

property Abc: Integer read **FAbc**;

или:

property Abc: Integer read **GetAbc**;

т.е. не иметь не только Сеттера, но и ключевого слова **write**. Аналогично, Свойства могут быть **только для записи**, но такое используется редко.

Инкапсуляция должна обеспечивать **сокрытие** внутренней структуры компонента и обеспечивать его **целостность**, не позволяя пользователю перевести хранимые в нем данные в недопустимое или противоречивое

состояние. Обычно для этого **Сеттеры** снабжают дополнительными проверками корректности ввода, например, проверкой допустимого диапазона значений.

6.8. Примеры инкапсуляции

Синхронизация картинки и содержимого

Прежде всего в разработанном нами ранее Компоненте «картинка» **расходится** с введенными значениями, ее обновление произойдет только когда мы **принудительно** вызовем процедуру **Redraw**. Для решения этой проблемы снова откроем Пакет и в секции «**implementation**» допишем соответствующую команду перерисовки в конце Конструктора:

```
constructor TControlXO.Create(AOwner: TComponent);
begin
  inherited;
  FCountX := 3;
  FCountY := 3;
  FCellSize := 50;
  FColorOfLines := clBlack;
  Redraw;
end;
```

Аналогично допишем эту команду в конце всех Сеттеров:

```
procedure TControlXO.SetCellSize(const Value: Integer);
begin
  FCellSize := Value;
  Redraw;
end;

procedure TControlXO.SetColorOfLines(const Value: TColor);
begin
  FColorOfLines := Value;
  Redraw;
end;

procedure TControlXO.SetCountX(const Value: Integer);
begin
  FCountX := Value;
  Redraw;
end;

procedure TControlXO.SetCountY(const Value: Integer);
begin
  FCountY := Value;
  Redraw;
end;
```

Теперь, если вернуться к Проекту, то линии сетки на Компоненте (рис. 6.34) будут отображаться не только после запуска программы (**F9**), но и сразу в **Design-time**. Более того, если сейчас изменить Свойства компонента через Инспектор объектов, то он сразу будет перерисован с новыми настройками.

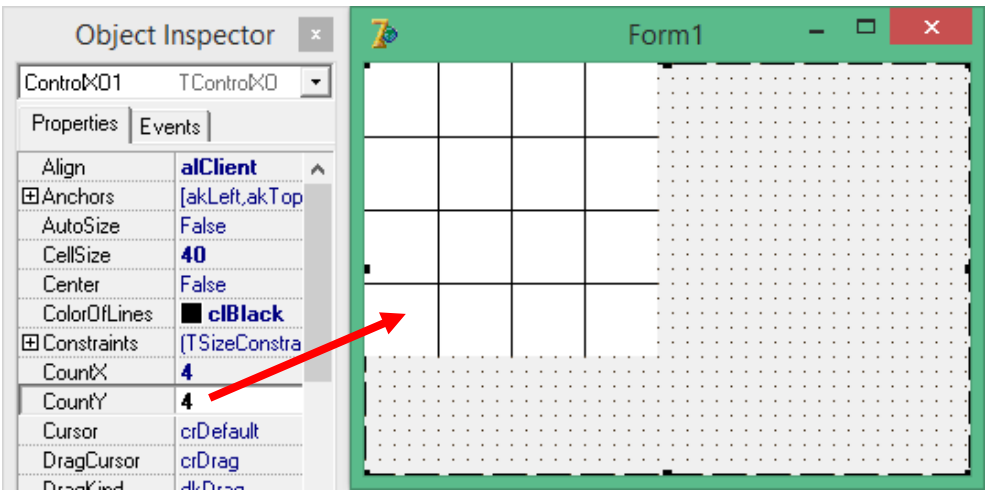


Рисунок 6.34 – Линии сетки Design-time

Проверка допустимого диапазона значений

Сейчас мы можем ввести через Инспектор объектов недопустимые значения, например, ширину клетки в 100 тыс. пикселей, или размер поля 0x0, или даже отрицательные значения. Для решения данной проблемы предназначены Сеттеры.

Для разработанного нами Компонента будем использовать следующие диапазоны значений:

- размер поля от **1x1** до **100x100**;
- размер клетки от **10** до **100** пикселей.

У нас есть два основных варианта действий при вводе значения, выходящего за допустимые границы:

- вообще не менять значение (т.е. оставить старое значение), например:

```
if (Value > 0) and (Value <= 100) then FCountX := Value;
```

- вывести вместо неправильного значения ближайшее допустимое (т.е. минимальное или максимальное значение):

```
FCellSize := Value;
if Value < 10 then FCellSize := 10;
if Value > 100 then FCellSize := 100;
```

6.9. Добавление собственных событий

События тоже являются Свойствами, поэтому они также объявляются через «**property**». Если мы хотим, чтобы Событие попало в Инспектор объектов, то его нужно объявлять в секции «**published**». События принято объявлять **после** остальных Свойств, т.е. в конце секции.

Объявим Событие, которое будет наступать после каждой перерисовки Компонента (после выполнения процедуры **Redraw**):

```
//События
property OnRedraw: TNotifyEvent;
```

Если после этого нажать «**волшебную**» комбинацию клавиш «**Ctrl+Shift+C**», то Свойство будет дополнено соответствующими Полем и Сеттером:

```
//События
property OnRedraw: TNotifyEvent read FOnRedraw write SetOnRedraw;
```

В Сеттере необходимо прописать команду **Redraw** *(как мы делали это ранее в **5** местах)*. Это необходимо, чтобы перерисовка наступала в момент присвоения Обработчика события:

```
procedure TControlXO.SetOnRedraw(const Value: TNotifyEvent);
begin
  FOnRedraw := Value;
  Redraw;
end;
```

Если сейчас посмотреть на этот Компонент в Инспекторе объектов, то в списке Событий у него появится **OnRedraw**. Но это Событие не наступит **никогда**, т.к. это **мы** должны сказать, когда оно наступит.

Для этого в конце реализации процедуры **Redraw** необходимо написать следующий код:

```
procedure TControlXO.Redraw;
  ...
  //Событие перерисовки
  if Assigned(FOnRedraw) then FOnRedraw(Self);
end;
```

где **FOnRedraw(Self)** – означает вызов соответствующего События;

Assigned(FOnRedraw) – означает проверку того, что Обработчик события назначен *(т.к. пользователь может и не использовать данное Событие в своей работе)*.

Теперь закроем Пакет и перейдем к Проекту. Для нашего нового События **OnRedraw** напишем код, который заполняет всё игровое поле «ноликами» (рис. 6.35):

```
//Событие перерисовки
procedure TForm1.ControlXO1Redraw(Sender: TObject);
var X, Y: Integer;
    X1, Y1, X2, Y2: Integer;
begin
  //Рисуем фигуры
  for X := 1 to ControlXO1.CountX do
  begin
    for Y := 1 to ControlXO1.CountY do
    begin
      //Координаты фигуры
      X1 := Round(ControlXO1.CellSize * (X-0.8));
      Y1 := Round(ControlXO1.CellSize * (Y-0.8));
      X2 := Round(ControlXO1.CellSize * (X-0.2));
      Y2 := Round(ControlXO1.CellSize * (Y-0.2));

      //Рисуем нолики
      ControlXO1.Picture.Bitmap.Canvas.Ellipse(X1, Y1, X2, Y2);
    end;
  end;
end;
```

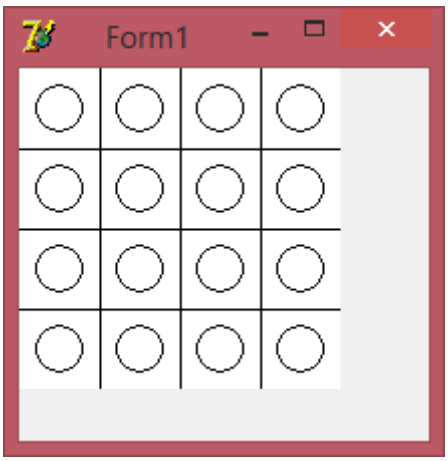


Рисунок 6.35 – Поле, заполненное «ноликами»

Аналогично добавим рисование «крестиков». Пока будем рисовать их прямо поверх ноликов (рис. 6.36):

```
//Рисуем крестик
ControlXO1.Picture.Bitmap.Canvas.MoveTo(X1, Y1);
ControlXO1.Picture.Bitmap.Canvas.LineTo(X2, Y2);
ControlXO1.Picture.Bitmap.Canvas.MoveTo(X1, Y2);
ControlXO1.Picture.Bitmap.Canvas.LineTo(X2, Y1);
```

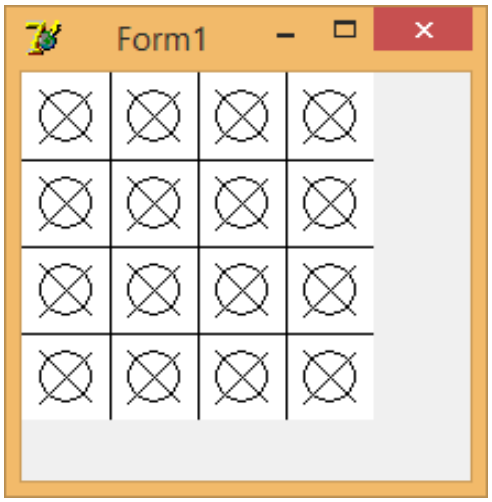


Рисунок 6.36 – Поле, заполненное «ноликами» и «крестиками»

6.10. Свойство-массив

Чтобы создать полноценную игру, нам нужно иметь Массив, хранящий текущее состояние игры. Это двумерный массив. Ноль будет означать отсутствие фигуры (т.е. пустую клетку), «1» – фигуру первого игрока (т.е. «крестик»), «-1» – фигуру второго игрока (т.е. «нолик»).

В дальнейшем это можно распространить и на другие игры, например, для шашек это могут быть «1» и «-1», соответственно для белых и черных фигур («простых»), а «2» и «-2» – для «дамок». Для шахмат это могут быть, например, «±1» – для пешек, «±2» – для коней, «±3» – для слонов и т.д.

Максимально возможный размер поля, который ранее мы ограничили Сеттерами, у нас составляет 100x100, поэтому мы будем использовать массив **100x100 фиксированной** длины. *Это не сильно увеличит размер занимаемой оперативной памяти.*

Массив объявляется как свойство с индексом (индексами). Свойство с индексом **нельзя** менять через Инспектор объектов, поэтому нужно объявить его в секции «**public**», а не «**published**»:

```
property Item[X, Y: Integer]: Integer;
```

После нажатия «**волшебной**» комбинации клавиш «**Ctrl+Shift+C**», мы увидим, что для «массива» всегда создаются как Сеттер, так и Геттер:

```
property Item[X, Y: Integer]: Integer read GetItem write SetItem;
```

Более того, если посмотреть повнимательнее, то можно заметить, что реализации Сеттера и Геттера здесь **пустые** *(в отличие от тех Свойств, что мы добавляли ранее)*. Кроме того, в этом случае отсутствует и Поле, начинающееся с буквы «**F**».

Нам придется самостоятельно объявить массив (в секции «**private**»):

```
FItem: Array[1..100, 1..100] of Integer;
```

Кроме того, нам придется вручную заполнить Геттер и Сеттер:

```
function TControlXO.GetItem(X, Y: Integer): Integer;
begin
  Result := FItem[X, Y];
end;

procedure TControlXO.SetItem(X, Y: Integer; const Value: Integer);
begin
  FItem[X, Y] := Value;
  Redraw;
end;
```

Конечно же если мы объявляем массив, то нам нужно добавить к Компоненту и процедуру (метод) **Clear** для его очистки.

```
procedure TControlXO.Clear;
var X, Y: Integer;
begin
  //Очистка поля
  for X := 1 to 100 do for Y := 1 to 100 do FItem[X, Y] := 0;
  Redraw;
end;
```

Вернемся в Проект и добавим **Кнопку**. Она будет расставлять несколько произвольных фигур на поле:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //Заполняем поле примерами значений
  ControlXO1.Item[2, 2] := 1;
  ControlXO1.Item[1, 2] := -1;
  ControlXO1.Item[2, 1] := 1;
end;
```

Также подправим Код, рисующий «крестики» и «нолики». Теперь он должен выполняться не для всех клеток, а только для тех, где в Массиве записаны «1» и «-1»:

```
//Рисуем нолики
if ControlXO1.Item[X, Y] = -1 then
begin
  ControlXO1.Picture.Bitmap.Canvas.Ellipse(X1, Y1, X2, Y2);
end;

//Рисуем крестик
if ControlXO1.Item[X, Y] = 1 then
begin
  ControlXO1.Picture.Bitmap.Canvas.MoveTo(X1, Y1);
  ControlXO1.Picture.Bitmap.Canvas.LineTo(X2, Y2);
  ControlXO1.Picture.Bitmap.Canvas.MoveTo(X1, Y2);
  ControlXO1.Picture.Bitmap.Canvas.LineTo(X2, Y1);
end;
```

Кроме того, нужно добавить в программу кнопку или пункт меню для очистки поля (**Clear**). В итоге после запуска (**F9**) и нажатия кнопки мы увидим следующее окно (рис. 6.37).

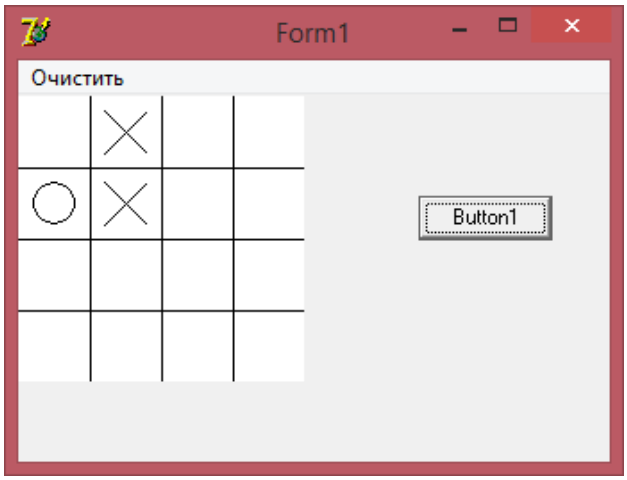


Рисунок 6.37 – Поле с несколькими фигурами

Теперь мы можем **переместить** весь код, написанный в обработчике **OnRedraw** вовнутрь Компонента, в процедуру **Redraw**. Кроме того, необходимо **добавить** к компоненту два новых свойства **ColorX** и **ColorO**, позволяющих менять цвета соответствующих фигур. Начальные значения цветов – **зеленый** и **синий**. *При желании можно также добавить свойства **WidthX** и **WidthO**, позволяющие менять толщину линий для фигур.*

6.11. *Свойство-массив по умолчанию

Для одного из массивов (одного из свойств с индексом) можно использовать более компактный способ записи. Так при обращении к массиву вместо:

ControlXO1.Item[X, Y]

достаточно будет писать:

ControlXO1[X, Y]

Для реализации такой возможности требуется дописать ключевое слово **default** к соответствующему свойству:

```
property Item[X, Y: Integer]: Integer read GetItem write SetItem; default;
```

Ключевое слово **default** применяется **исключительно к массивам** *(для других свойств это ключевое слово имеет совершенно иное значение)*. Только **одно** свойство можно использовать как свойство по умолчанию *(в более новых версиях Delphi, можно использовать и несколько свойств по умолчанию, но все они должны иметь одно имя)*. Размерность массива, т.е. количество индексов, может быть любым.

Например, для класса **TStrings** в системе уже имеется свойство-массив по умолчанию Strings [Index: Integer]. Так для получения первой строки текста, введенного в компонент RichEdit1, необходимо написать:

S := RichEdit1.Lines.Strings[0];

Но т.к. это массив по умолчанию, то для получения того же результата достаточно использовать:

S := RichEdit1.Lines[0];

Именно второй способ чаще всего применяется для Lines компонентов RichEdit и Memo.

6.12. Действия мышки

Разработанный нами Компонент уже много что умеет, но для игр важно **взаимодействие** с пользователем. В данном случае «крестики» и «нолики» должны расставляться на поле при помощи **мышки**.

За нажатие мышки отвечает событие **OnMouseDown**. Но это событие возвращает координаты **X** и **Y** в **пикселях**, а нам необходимо знать **номер ячейки**, на которой был произведен клик мышкой. Для этого необходимо использовать целочисленное деление.

Временно будем отображать эти новые координаты в заголовке Формы (рис. 6.38):

```
procedure TForm1.ControlXO1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Пересчет координат в номер ячейки
  X := (X div ControlXO1.CellSize);
  Y := (Y div ControlXO1.CellSize);

  //Выводим номер ячейки
  Caption := IntToStr(X) + ', ' + IntToStr(Y);
end;
```

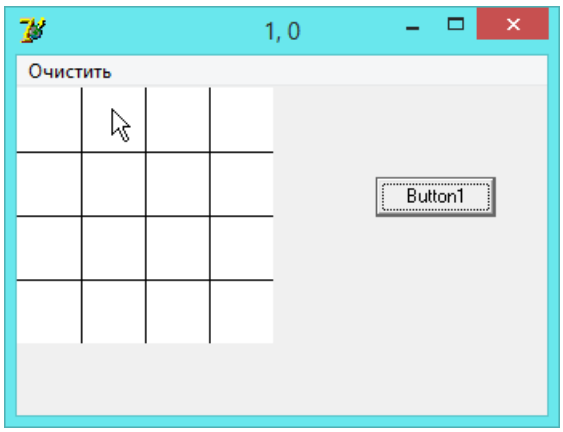


Рисунок 6.38 – Координаты нажатия в заголовке

Данные координаты начинаются с «**нуля**», но мы хотим нумеровать ячейки, начиная с «**единицы**», поэтому допишем к ним «**+1**».

В ячейку, по которой мы кликнули, будем ставить «**крестик**»:

```
procedure TForm1.ControlXO1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Пересчет координат в номер ячейки
  X := (X div ControlXO1.CellSize) + 1;
  Y := (Y div ControlXO1.CellSize) + 1;

  //Поставить крестик (если клетка свободна)
  if ControlXO1.Item[X,Y] = 0 then ControlXO1.Item[X,Y] := 1;
end;
```

Причем мы ставим «крестик» только в пустую клетку («**=0**»).

Если заменить значение «1» на «-1», то можно ставить «**нолики**» вместо «крестиков»:

```
//Поставить нолик (если клетка свободна)
if ControlXO1.Item[X,Y] = 0 then ControlXO1.Item[X,Y] := -1;
```

Но мы хотим, чтобы «крестики» и «нолики» ставились **по очереди**. Для этого необходимо добавить **глобальную** переменную **NextStep**, хранящую информацию о том, кто ходит следующим. Значение «1» будет означать, что ходит «Крестик», «-1» – «нолик».

```
var NextStep: Integer = 1;

procedure TForm1.ControlXO1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Пересчет координат в номер ячейки
  X := (X div ControlXO1.CellSize) + 1;
  Y := (Y div ControlXO1.CellSize) + 1;

  //Поставить фигуру (если клетка свободна)
  if ControlXO1.Item[X,Y] = 0 then
  begin
    ControlXO1.Item[X,Y] := NextStep;
    //Передать ход другому
    NextStep := -NextStep;
  end;
end;
```

Если сейчас начинать **новую** игру, используя пункт меню «**Очистить**», то первым может ходить как «крестик», так и «нолик». При очистке поля необходимо **сбрасывать** и переменную **NextStep** так, чтобы первым ходил «крестик»:

```
procedure TForm1.N1Click(Sender: TObject);
begin
  //Очистка поля
  ControlXO1.Clear;
  NextStep := 1;
end;
```

6.13. *Перехват событий

На данный момент код для пересчета координат расположен в Проекте. Но правильнее будет перехватывать событие OnMouseDown и генерировать вместо него собственное событие OnMouseCell, возвращающее не координаты в пикселях, а номера ячеек. Таким образом мы переместим код для пересчета координат из Проекта в Компонент;

Добавление события

Зайдем в Пакет и добавим к Компоненту еще одно Событие:

```
property OnClickCell: TMouseMoveEvent;
```

!!! Это событие можно объявить и без Сеттера, как:

```
property OnClickCell: TMouseMoveEvent read FOnClickCell write FOnClickCell;
```

Написание кода

Сейчас данное событие никогда **не** наступит, но тем не менее пока закроем Пакет и откроем Проект. Для нового События **OnClickCell** создадим обработчик:

```
procedure TForm1.ControlXO1ClickCell(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin

end;
```

В этот обработчик перенесем вторую часть кода из обработчика События **OnMouseDown**. Это та часть, которая касается фигур и Массива. В итоге получим:

```
procedure TForm1.ControlXO1ClickCell(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  //Поставить фигуру (если клетка свободна)
  if ControlXO1.Item[X,Y] = 0 then
  begin
    ControlXO1.Item[X,Y] := NextStep;
    //Передать ход другому
    NextStep := -NextStep;
  end;
end;
```

В обработчике События **OnMouseDown** (на месте вырезанного кода) необходимо вызвать наступление нашего нового события, т.е. в итоге получим код:

```
procedure TForm1.ControlXO1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Пересчет координат в номер ячейки
  X := (X div ControlXO1.CellSize) + 1;
  Y := (Y div ControlXO1.CellSize) + 1;

  //Вызов события
  if Assigned(ControlXO1.OnClickCell) then
    ControlXO1.OnClickCell(Self, Shift, X, Y);
end;
```

Перемещение обработчика события в компонент

Теперь код разбит на две части, и пересчет координат можно вынести в Компонент *(только пока непонятно, в какое именно место?..)*.

В секции **public** Компонента необходимо объявить в точности следующую процедуру:

```
procedure MouseDown(Button: TMouseButton; Shift: TShiftState;
                    X, Y: Integer); override;
```

Она очень похожа на обработчик, который мы хотим заменить:

```
procedure TForm1.ControlXO1MouseDown(Sender: TObject;
                                     Button: TMouseButton;
                                     Shift: TShiftState;
                                     X, Y: Integer);
```

Далее нажимаем **Ctrl+Shift+C** и получаем:

```
procedure TControlXO.MouseDown(Button: TMouseButton;
                               Shift: TShiftState; X, Y: Integer);
begin
    inherited;
    ...
end;
```

Именно сюда и нужно переместить код из Проекта.

6.14. Абстракция

Абстракция в объектно-ориентированном программировании – это использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе. Основная идея состоит в том, чтобы представить объект минимальным набором **полей** и **методов** и при этом с достаточной точностью для решаемой задачи.

Абстракция является одной из основ ООП и позволяет работать с объектами, **не** вдаваясь в особенности их реализации.

Термин «**Абстракция**» появился гораздо раньше, чем Объектно-ориентированное программирование. Например, когда для «Калькулятора» мы выделяли часть функций в отдельный модуль – это тоже была форма абстракции. *Мы будем рассматривать абстракцию в программировании вообще, а не только в ООП.*

Уровень абстракции – один из способов **сокрытия** рабочих деталей подсистемы. Применяется для управления сложностью проектируемой системы при **декомпозиции**. В этом случае система представляется в виде **иерархии** уровней абстракции.

Декомпозиция – разделение целого на части. Про декомпозицию писал ещё Декарт (задолго до появления компьютеров).

Примером программного обеспечения, использующего уровни абстракции, является **семиуровневая модель OSI**. Каждый уровень модели **OSI** рассматривает **отдельную** часть требований по организации связи, сокращая таким образом сложность соответствующих инженерных решений.

Абстракция и декомпозиция применимы не только для программирования, но и вообще для **любой** техники. Так компьютер состоит из четко выраженных элементов в виде процессора, оперативной памяти, материнской платы, жесткого диска, монитора и т.п. Каждый из которых может изготавливаться различными производителями и по различным технологиям, но все они будут (в теории) совместимы между собой благодаря стандартизированным интерфейсам.

Знания, передаваемые студентам в Университетах, тоже были подвергнуты **декомпозиции** – т.е. разбиты на разные предметы с различными названиями.

Уровень абстракции определяется не только для всего Проекта, но и для отдельных его частей и компонентов. *Так в одном проекте часть кода может иметь избыток абстракции, а часть – недостаток.*

Как гласит старинная шутка: «Любую архитектурную проблему можно решить добавлением дополнительного слоя абстракции» («кроме проблемы большого количества абстракций»).

В нашем случае

Для разрабатываемой нами игры в Крестики-нолики, мы **вынесли** бОльшую часть данных и кода в отдельный Компонент (а также в отдельный Модуль), разбив нашу программу на **два** уровня. Так за всю отрисовку, вычисление координат и хранение данных и настроек отвечает Компонент. А реализация правил игры, порядок ходов, определение окончания игры – должны быть реализованы в **отдельном** Проекте, который использует наш Компонент. *Мы занимались в основном **первой** частью (т.е. Компонентом), а вторая часть (т.е. Проект) осталась практически **пустой** – что и хорошо. Она может быть реализована студентами **самостоятельно**.*

6.15. РАБОТА № 4

«Создание визуального компонента»

Разработать Компонент, производящий отрисовку поля для игры в крестики-нолики, а также реализовать Проект, демонстрирующий его работу. При выполнении работы руководствоваться принципами абстракции, наследования и инкапсуляции.

Компонент должен содержать:

- свойства, задающие размеры поля (по умолчанию 3x3), размеры клеток, цвет сетки и цвета фигур каждого из игроков;
- начальные значения для добавленных свойств и ограничения диапазонов вводимых в них значений;
- перерисовку (Redraw) Компонента, выполняемую автоматически при каждом изменении свойств;
- свойство-массив, хранящее текущее состояние игры *(достаточно статического массива)*;
- возможность начать новую игру (метод Clear или New, очищающий поле);
- хотя бы одно новое (собственное) событие.

Проект должен содержать:

- разработанный Компонент;
- кнопку или пункт меню «Новая игра»;
- возможность поочередно осуществлять ходы щелчком мышки по выбранной клетке (первым всегда ходит крестик).

!!! В этой версии конец игры определять не обязательно. Ходы можно осуществлять до тех пор, пока не будут заняты все клетки.

6.16. *РАБОТА № 4 (дополнения) «Игра Крестики-нолики»

Дополнить предыдущую версию программы следующим функционалом:

- возможность изменять размеры поля и ширину клетки во время игры (когда программа запущена);
- дать пользователю возможность изменять настройки цветов сетки и фигур;
- возможность изменять толщину линий рисуемых фигур;
- ограничить возможность клика мышкой за пределами видимой сетки;
- определять конец игры (когда 3 фигуры выстроены в ряд) и не давать совершить при этом следующий ход; рисовать «финальную линию», показывающую, какие именно 3 фигуры выстроены в ряд; добавить возможность выбора цвета и толщины финальной линии через Инспектор объектов;
- добавить пункты меню «О программе» и/или «Справка», в которых в отдельном окне рассказывается о разработчике игры, её правилах и возможностях;
- *перехватывать событие OnMouseDown и генерировать вместо него собственное событие OnMouseCell, но возвращающее не координаты в пикселях, а номера ячеек, *переместив, таким образом, код для пересчета координат из Проекта в Компонент*;
- *выпустить специальную версию игры «крестик, нолик и квадрат» с тремя игроками, ходящими по очереди;
- **заменить статический массив на динамический;

- **руководствуясь знаниями, полученными при создании «Крестиков-ноликов», создать поле для игры в шашки, шахматы, сапёра или др. подобные игры.

6.17. *Динамическое создание экземпляра класса

Обычно в Delphi используется **статическое** создание Компонентов, для этого мы просто добавляем их из Палитры компонентов на форму, а всё остальное за нас делает Delphi. В этом случае все Компоненты создаются сразу вместе с формой *(например, при запуске Приложения)*, пока форма существует – существуют и эти компоненты, их уничтожение происходит вместе с уничтожением формы *(например, при закрытии Приложения)*.

Но можно создавать любые из Компонентов/Объектов и **динамически**, из кода программы. В этом случае Компоненты не обязаны существовать всё время *(пока работает Приложение)*. Мы сами контролируем, когда создавать Компонент, а когда его уничтожить.

В общем случае, для использования динамического Компонента (Объекта) требуется следующее:

- объявить переменную соответствующего типа;
- создать Компонент при помощи конструктора **Create**;
- произвести настройку Компонента;
- использовать Компонент для работы;
- уничтожить Компонент, выполнив команду **Free**.

!!! Динамическое создание Компонентов (Объектов) опасно возможностью возникновения «утечек памяти», при неправильном их уничтожении (подробнее см. раздел **6.18**), или появления ошибок («Исключений», «Exception») в случае, если компонент не создан или уже не существует (см. рис. 6.43).

Стоит выделить 3 случая создания Объектов/Компонентов, т.к. механизмы работы с ними будут различаться:

1. Объекты. Здесь имеются в виду Объекты в узком смысле («обычные Объекты»), без Компонентов (визуальных и невизуальных);
2. Компоненты. В узком смысле, без визуальных Компонентов (Control), либо визуальные компоненты, в случае, когда их «визуальность» никак не проявляется, т.е. их отображение не планируется;
3. Контролы (визуальные компоненты). В случае, если они отображаются на форме.

В данном разделе речь идет прежде всего о «невидимых» компонентах (не путать с невизуальными!), которые хранятся только в оперативной памяти компьютера и не отображаются для пользователя. Про «видимые» на форме компоненты, см. раздел **6.19**.

Динамическое создание объектов в памяти

Для создания нового Объекта в памяти требуется выполнить как минимум 3 следующие действия:

1. объявить переменную, например:

```
var Bmp0: TBitmap;
```

2. создать Объект командой Create:

```
Bmp0 := TBitmap.Create;
```

3. уничтожить Объект (после его использования):

```
Bmp0.Free;
```

Кроме того, механизм работы с Объектом (и его уничтожение) зависит от того, объявлен он как Локальная переменная или как Глобальная.

Локальная переменная

В случае использования Локальной переменной для Объекта он должен быть уничтожен в той же процедуре/функции, в которой создан, например:

```
procedure TForm1.Button1Click(Sender: TObject);
var Bmp0: TBitmap;
begin
  ...
  Bmp0 := TBitmap.Create;
  ...
  Bmp0.Free;
  ...
end;
```

Правильно будет оформить такой код через конструкцию **try-finally**:

```
procedure TForm1.Button1Click(Sender: TObject);
var Bmp0: TBitmap;
begin
  ...
  Bmp0 := TBitmap.Create;
  try
    ...
  finally
    Bmp0.Free;
  end;
  ...
end;
```

!!! Весь код для работы с Объектом должен быть расположен между **try** и **finally**.

Блок **finally** гарантирует, что Объект будет уничтожен в любом случае, даже если при выполнении кода на пути от **try** к **finally** произойдет ошибка. *В случае если ошибка не возникает, код будет выполняться так, как если бы соответствующих ключевых слов **try**, **finally** и **end** не было вовсе.*

Сам такой Объект правильно будет называть даже не «локальным», а «**временным**», т.к. он создается для выполнения конкретной задачи, после чего сразу уничтожается за ненадобностью. *Тогда остальные Объекты в этом контексте можно будет назвать не «глобальными», а «долгоживущими».*

Глобальная переменная

Случай с Глобальной переменной для Объекта отличается тем, что создание и уничтожение происходят в разных местах (в разных процедурах, функциях или т.п.). *Если бы создание и уничтожение происходило в одном месте, то не было бы и смысла в Глобальной переменной, достаточно бы было Локальной.*

Например:

```
var Bmp0: TBitmap;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Bmp0 := TBitmap.Create;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  Bmp0.Free;
end;
```

Соответственно, здесь мы уже не сможем использовать конструкцию **try-finally**. Но уничтожать Объект мы все еще обязаны, что мы и делаем в момент уничтожения формы (**OnDestroy**) – в случае, если Объект используется формой (например, создан в **OnCreate** этой формы), а связанная с ним переменная, объявлена в Модуле этой формы. *Либо уничтожаем Объект в деструкторе **Destroy** нашего Компонента, в случае если Объект создается в этом Компоненте (например, в его конструкторе **Create**), а связанная с ним переменная – это Поле нашего Компонента. Либо в блоке **finalization** Модуля (**Unit**), в случае, когда Объект был создан в этом Модуле (например, в его блоке **initialization**), а связанная с ним переменная, является переменной этого Модуля.*

** Динамическое создание компонентов в памяти

Компонент (визуальный или невизуальный) отличается от Объекта наличием владельца (**Owner**), который указывается в конструкторе при создании компонента. Например, в данном случае владельцем будет Form1:

```
var HTTP: TIdHTTP;
...
HTTP := TIdHTTP.Create(Form1);
```

В свою очередь, вариант с Компонентами подразделяется еще на несколько подвариантов.

1. «**Долгоживущий**» – для компонента объявлена Глобальная переменная и задан владелец (например, Form1):

```
var HTTP: TIdHTTP;

procedure TForm1.FormCreate(Sender: TObject);
begin
  HTTP := TIdHTTP.Create(Form1);
end;
```

В данном случае **нет** необходимости в явном уничтожении Компонента командой HTTP.**Free**, т.к. за уничтожение компонента будет отвечать его

владелец. *Хотя использование команды **Free** и возможно в любой момент по желанию разработчика.*

!!! Как правило, в качестве владельца может выступать **Self**, который означает ссылку «**на себя**». Т.е. на текущий Компонент (Объект), внутри которого мы находимся, например, если мы пишем код для формы TForm1.FormCreate, то мы находимся внутри Form1, а если пишем код для нашего Компонента TControlXO.Redraw, то **Self** будет ссылаться на ControlXO1 (или другой текущий экземпляр этого класса, такой как ControlXO2, ControlXO3 или т.п.).

Такая переменная может быть объявлена в любом возможном месте: в Модуле формы, в другом Модуле, как Поле нашего Компонента, как Поле формы, как переменная Процедуры/Функции и т.п.

!!! На самом деле, в данном случае переменная может быть и Локальной – тогда компонент продолжит существовать и когда переменной уже не будет (но и обратиться к нему через эту переменную станет невозможно). *Переменная может не быть объявлена вовсе.*

2. «**Без владельца**» (также является «долгоживущим») – для компонента объявлена Глобальная переменная, но владелец не указан (**nil**):

```
var HTTP: TIdHTTP;

procedure TForm1.FormCreate(Sender: TObject);
begin
  HTTP := TIdHTTP.Create(nil);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  HTTP.Free;
end;
```

В данном случае использование команды **Free** является обязательным, а переменная **не** может быть Локальной! Этот вариант полностью аналогичен варианту для Объектов с Глобальными переменными, описанному ранее. *Фактически, здесь Компонент используется не как Компонент, а как Объект (в узком смысле).*

Уничтожать такой Компонент нужно в **OnDestroy** формы – если связанная с ним переменная объявлена в Модуле этой формы; в деструкторе **Destroy** нашего Компонента – если связанная с ним переменная является Полем нашего Компонента; либо в блоке **finalization** Модуля (**Unit**) – если переменная является переменной этого Модуля.

!!! В любом случае у разработчика всегда есть возможность применить **Free** и раньше, не дожидаясь **OnDestroy**, **Destroy** или **finalization**.

!!! У Компонентов может быть и несколько конструкторов **Create** с разным количеством и типом аргументов. В частности, у рассматриваемого в примере TIdHTTP есть конструктор **без** параметров (как у обычных Объектов), который создает Компонент без владельца (**Owner** = **nil**), т.е. для него можно написать:

```
HTTP := TIdHTTP.Create;
```

но это всё ещё тот же случай без владельца.

3. «**Временный**» – аналогичен варианту для «временных» Объектов, описанному ранее. В нем обязательно использование **Free** и конструкции **try-finally**:

```
procedure TForm1.Button1Click(Sender: TObject);
var HTTP: TIdHTTP;
begin
  HTTP := TIdHTTP.Create(nil);
  try
    Memo1.Text := HTTP.Get('http://188.134.16.135/');
  finally
    HTTP.Free;
  end;
end;
```

Данный случай не столь привязан к владельцу как предыдущие, т.е. в примере могло быть написано и Form1 или **Self** вместо **nil**.

!!! Весьма опасным является случай, когда мы не используем **Free** явно (либо не используем для него **finally**) и при этом владелец для данного Компонента **задан**, а компонент далее нигде **не** используется. Тогда Компонент будет освобожден **позднее**, чем это нужно было сделать (часто в Диспетчере задач можно наблюдать увеличение занимаемой Приложением памяти).

Подобную утечку памяти даже не отловить стандартными средствами Delphi (описанными далее в разделе **6.18**), т.к. формально, Компонент уничтожается до завершения программы (просто не в своё время), а система распознает его как подвариант **1**, а не как подвариант **3** (т.е. как «долгоживущий», а не как «временный»).

!!! Учитывая сказанное, для заведомо временных компонентов лучше всегда использовать в качестве владельца **nil**. Тогда можно будет отловить данный вид утечек памяти при отладке (подробнее см. раздел **6.18**).

6.18. **Утечки памяти

В Delphi **7** нет такой функции, но в **новых** версиях Delphi можно активировать подсчет неуничтоженных компонентов. Для этого используется команда:

ReportMemoryLeaksOnShutdown := True;

Теперь при закрытии Приложения при наличии неуничтоженных компонентов отобразится окно с их списком (рис. 6.39).

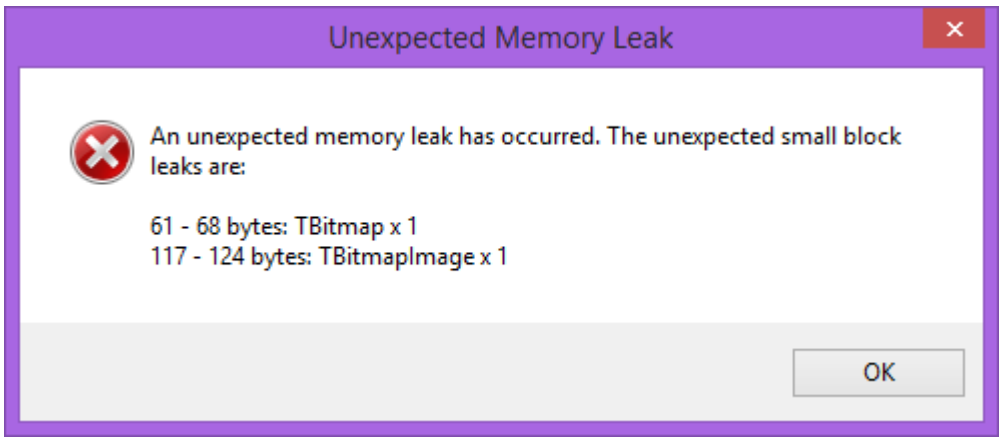


Рисунок 6.39 – Утечка памяти

Независимо от того, отображаем мы эту информацию при закрытии Приложения или нет, неуничтоженные Компоненты приводят к «утечкам памяти», когда неиспользуемые более Компоненты, продолжают занимать оперативную память компьютера.

Худшим вариантом утечек является **постоянное** создание новых Компонентов (например, в цикле) без их уничтожения. Данный вид утечки можно отследить даже через обычные Диспетчер задач (**Ctrl+Shift+Esc**). Размер программы (в памяти) будет постоянно расти (хотя возможно и медленно), пока не займет всю предоставленную память.

Где использовать команду?

Объявить данную команду можно в самом начале **OnCreate** главной формы. Но еще лучше будет активировать ее в начале главного файла Приложения (***.dpr**), сразу после **begin** и до **Initialize**.

Как использовать команду?

Правильно будет обернуть данную запись соответствующими директивами компилятора:

```
{$IFDEF DEBUG}ReportMemoryLeaksOnShutdown := True;{$ENDIF}
```

В этом случае, в **новых** версиях Delphi она будет работать только в режиме отладки **Debug**, и результат ее работы никогда не увидит конечный пользователь, для которого компилируется версия **Release**. Кроме того, при использовании этого кода в **старых** версиях Delphi, подобная запись не приведет к ошибкам компиляции. *Правда и работать в старых версиях она не будет.*

6.19. *Динамическое создание компонентов на форме

Компоненты на форме являются Объектами, поэтому они проходят тот же жизненный цикл, что и любые Объекты (см. раздел **6.17**). Но вот с точки зрения написания программного кода для их динамического создания, имеются значительные отличия. Для размещения нового Компонента на форме необходимо:

- создать Компонент при помощи конструктора **Create**;
- настроить владельца (**Owner**) и родителя (**Parent**) для Компонента;
- произвести настройку внешнего вида Компонента, его размеров и положения, надписи на компоненте и т.п.;
- привязать к Компоненту события, если они есть (например, если необходим клик мышкой на Компоненте).

Дополнительно (но не обязательно!) можно:

- объявить для Компонента переменную (локальную или глобальную);
- досрочно уничтожить Компонент, выполнив команду **Free**.

Например, напишем код для кнопки, создающей другую кнопку:

```
procedure TForm1.Button1Click(Sender: TObject);
var NewButton: TButton;
begin
  //Создание кнопки
  NewButton := TButton.Create(Form1);
  NewButton.Parent := Form1;
  //Размеры, положение и надпись на кнопке
  NewButton.Height := 25;
  NewButton.Width := 100;
  NewButton.Left := 20;
  NewButton.Top := 100;
  NewButton.Caption := 'Новая кнопка';
  //События кнопки
  NewButton.OnClick := NewButtonClick;

  //!!! В конце не нужно выполнять NewButton.Free !!!
end;
```

!!! Т.к. мы уже находимся внутри TForm1, то в обоих местах в коде вместо «**Form1**» достаточно было написать «**Self**». *Этот вариант был бы даже более предпочтительным.*

В коде мы привязываем к нажатию кнопки **OnClick** обработчик события «**NewButtonClick**», поэтому также нам нужно объявить этот обработчик. В данном случае кнопка просто будет самоуничтожать сама себя при нажатии на неё:

```
procedure TForm1.NewButtonClick(Sender: TObject);
begin
  //Самоуничтожение кнопки
  Sender.Free;
end;
```

При создании нового Компонента, мы объявляем для него владельца (**Owner**) в конструкторе **Create**. Обычно владельцем выступает форма, для которой создается Компонент. Родителем (**Parent**) также выступает форма либо любая другая Панель на этой форме. Положение (Left и Top) нашего нового компонента отсчитывается в координатах родителя.

!!! Нам необходимо задать оба этих свойства.

Owner – это тот, кто отвечает за дальнейшее уничтожение Компонента. Это свойство есть у всех Компонентов (но не у Объектов!). И в качестве владельца может выступать любой Компонент (но не Объект!), в том числе и невизуальный компонент.

Parent – это то место, где будет отображен Компонент. Это свойство гораздо уже, оно есть только у визуальных компонентов (**Control**), а в качестве родителя может выступать только другой визуальный компонент (и то не каждый).

Т.к. при создании Компонента у него указан владелец (**Owner** = **Form1**), то этот владелец и будет уничтожать данный Компонент автоматически (при уничтожении формы и/или закрытии Приложения). Нам не понадобится явно использовать команду **Free** *(хотя она и будет выполнена, но не нами, а владельцем)*. При этом нам не запрещено в любой момент самостоятельно выполнить **Free** для досрочного уничтожения кнопки.

6.20. **Свойство-привязка

Можно добавить к Компоненту не только обычные Свойства (см. раздел **6.1**), Свойства-массивы (см. раздел **6.10**) или События (см. раздел **6.9**), но и свойства с привязкой к компоненту.

!!! В Инспекторе объектов такие свойства будут отображаться красным (как, например, свойства Action или PopupMenu).

Добавим к нашему Компоненту возможность привязки невизуального компонента с диалогом. Это будет диалог сохранения, используя который мы сможем запоминать текущее состояние игрового поля в виде файла с изображением.

Для использования диалога необходимо вначале подключить соответствующую библиотеку:

```
uses Dialogs;
```

Далее добавляем к компоненту новое свойство, и процедуру для работы с ним:

```
public
  procedure SaveAs;
published
  property SaveDialog: TSaveDialog;
end;
```

После нажатия комбинации **Ctrl+Shift+C**, описание свойства будет расширено до:

```
property SaveDialog: TSaveDialog read FSaveDialog write SetSaveDialog;
```

Остается только написать код для сохранения в файл:

```
procedure TControlXO.SaveAs;
begin
  if SaveDialog.Execute then
  begin
    Picture.SaveToFile(SaveDialog.FileName);
  end;
end;
```

Если теперь переключиться на Проект, использующий наш Компонент, то в Инспекторе объектов для него отобразится новое Свойство-привязка (рис. 6.40). В выпадающем списке этого Свойства можно увидеть все имеющиеся на форме диалоги сохранения (если они были добавлены на форму).

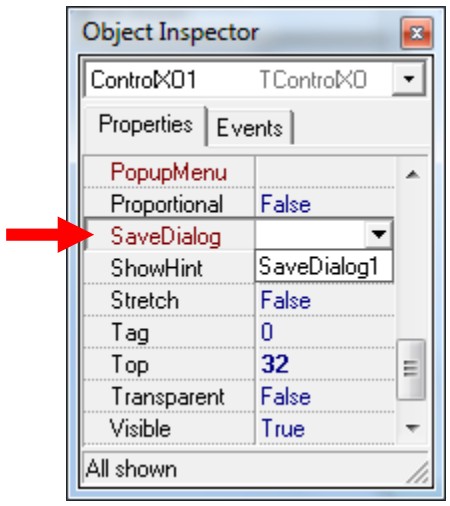


Рисунок 6.40 – Свойство с привязкой к компоненту

После привязки диалога к данному Свойству перед его именем появится знак «+» (рис. 6.41). Если теперь раскрыть этот знак «+», то можно настраивать диалог прямо внутри нашего компонента (рис. 6.42).

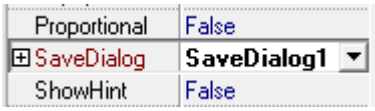


Рисунок 6.41 – Привязанный компонент

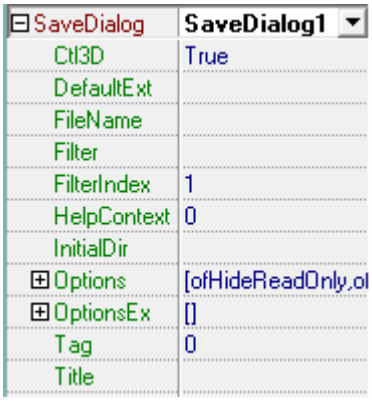


Рисунок 6.42 – Раскрытый компонент

Остается только добавить к нашей программе пункт меню (или кнопку) «Сохранить изображение», в обработчике которого написать:

```
ControlXO1.SaveAs;
```

Возможные ошибки

Если не привязать (см. рис. 6.40) к новому свойству никакого диалога, то при попытке обратиться к такому несуществующему диалогу (например, при вызове процедуры ControlXO1.SaveAs), мы получим ошибку (рис. 6.43).

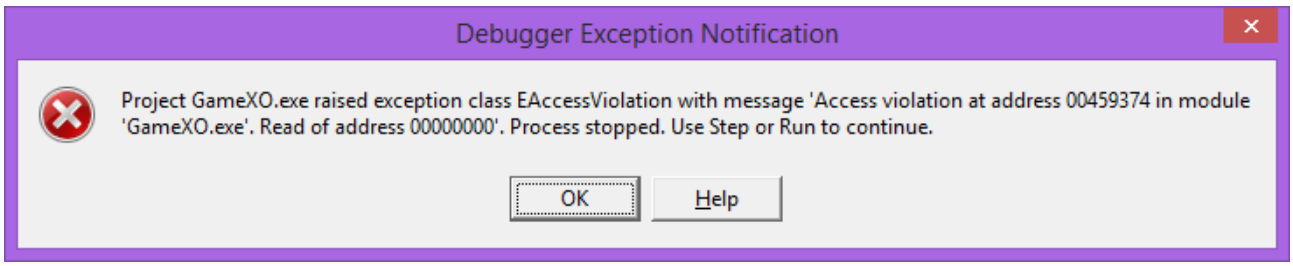


Рисунок 6.43 – Сообщение об ошибке

Для решения этой проблемы в коде необходимо использовать проверку следующего вида:

```
if Assigned(SaveDialog) …
```

Например, можно исправить сохранение файла следующим образом:

```
procedure TControlXO.SaveAs;
begin
  if Assigned(SaveDialog) and SaveDialog.Execute then
  begin
    Picture.SaveToFile(SaveDialog.FileName);
  end;
end;
```

!!! Данное изменение позволит избавиться от сообщения, но, на самом деле, не решит проблему, т.к. теперь при выборе соответствующего пункта меню не будет происходить ничего. *Хотя имело бы смысл, совсем скрывать такой недоступный пункт меню, или хотя бы делать его «серым» (недоступным).*

!!! Для рассмотренного примера лучше объявить Свойство не как Привязку, а как Подкомпонент, тогда при правильном использовании подобная ошибка не сможет появиться, а диалог всегда будет доступен.

6.21. **Свойство-компонент (Подкомпонент)

Данный вариант очень похож на предыдущий (см. раздел «6.20. **Свойство-привязка»), но стоит учесть следующее:

- данное Свойство имеет только **read** (без **write**);
- мы должны создавать этот Подкомпонент самостоятельно, т.е. динамически;
- нам обязательно понадобится использовать Конструктор (для главного компонента);
- владельцем Подкомпонента должен являться наш главный компонент, т.е. «**Self**».
- вложенный Компонент необходимо настроить как Подкомпонент, командой **SetSubComponent**, *иначе его свойства не будут сохраняться в файле *.dfm*;
- в отличие от предыдущего варианта, можно сразу задать из Кода начальные значения для Подкомпонента (например, имя файла для диалога);

- у данного Свойства больше не будет выпадающего списка с диалогами, нельзя будет перепривязать его к другому диалогу или отвязать от существующего диалога;
- теперь для работы нашего компонента не нужно будет добавлять какие-либо другие компоненты (например, диалоги) на форму.

В итоге нам понадобится написать следующий код:

```
uses Dialogs;

...

public
  constructor Create(AOwner: TComponent); override;
  procedure SaveAs;
published
  property SaveDialog: TSaveDialog read FSaveDialog;
end;

...

constructor TControlXO.Create(AOwner: TComponent);
begin
  inherited;
  FSaveDialog := TSaveDialog.Create(Self);
  FSaveDialog.SetSubComponent(True);
  FSaveDialog.Options := FSaveDialog.Options + [ofOverwritePrompt];
  FSaveDialog.Filter := 'Bitmaps (*.bmp)|*.bmp|All Files (*.*)|*.*';
  FSaveDialog.FileName := 'GameXO.bmp';
end;

procedure TControlXO.SaveAs;
begin
  if SaveDialog.Execute then
  begin
    Picture.SaveToFile(SaveDialog.FileName);
  end;
end;
```

Уничтожение компонента

Обычно уничтожать компонент должен тот, кто его создал. Здесь все так же, но для этого нам не нужно предпринимать никаких активных действий. Т.к. при создании Подкомпонента у него указан владелец («**Self**»), то этот владелец и будет уничтожать Подкомпонент автоматически. Нам не понадобится явно использовать Деструктор или **Free**.

!!! В данном случае мы **никогда** не должны использовать в нашем коде принудительный **Free** для этого Подкомпонента. Иначе это приведет к той же ошибке, что описана в разделе **6.20** (см. рис. 6.43).

КОНТРОЛЬНЫЕ ВОПРОСЫ ПО DELPHI И PASCAL

1. Язык Pascal. Классификация языков.
2. Операции и операторы в Pascal.
3. Логические и побитовые операции. Таблицы истинности. Примеры.
4. Комментарии. Примеры комментариев с кодом из разных разделов.
5. Объявление переменных и констант. Локальные и глобальные переменные. Примеры.
6. Преобразование типов. Примеры.
7. Математические функции. Примеры.
8. Строковые функции. Примеры использования.
9. Создание собственных процедур и функций. Аргументы. Примеры.
10. Модули Pascal. Структура. Создание модуля. Пример. Подключение модуля.
11. Условный оператор if. Варианты записи. Примеры. Блок-схемы для условий.
12. Цикл for. Варианты записи. Примеры.
13. Основные типы данных Delphi.
14. Объявление собственных типов.
15. Парадигмы программирования.
16. Императивные и Декларативные языки.
17. Среда разработки Delphi. RAD. IDE. Расширения файлов Delphi.
18. Библиотека VCL. Визуальные и невизуальные компоненты и их примеры. Design-time и Run-time. GUI.
19. Основные свойства визуальных компонентов. Свойства, задающие надписи и изображения на компонентах.
20. Свойства-привязки. Основные свойства формы.
21. Механизм Actions. Случаи использования Action. Свойства Action. Перечислить примеры стандартных Action.
22. Событийно-ориентированное программирование. Основные события визуальных компонентов. Примеры использования (с кодом).
23. Основные события невизуальных компонентов. Основные события формы. Примеры использования (с кодом).
24. Обзор визуальных компонентов Delphi: рассказать про группы компонентов «Метки и поля ввода», «Кнопки и галочки», «Ввод и отображение числовых значений». Общее, отличия.
25. Обзор визуальных компонентов Delphi: рассказать про группы компонентов «Списки», «Выпадающие списки», «Панели». Общее, отличия.
26. Свойства и методы полей Edit, Memo и RichEdit.
27. Диалоговые окна Application.MessageBox.

28. Рисование графических объектов. Цвета и стили рисования. Двойная буферизация.
29. Объектно-ориентированное программирование. Основные термины ООП. Методы. Поля. Свойства. Конструкторы и деструкторы. Пример объявления свойств компонента и их начальных значений.
30. Наследование. Термины и определения, используемые в наследовании.
31. Инкапсуляция. Термины и определения, используемые в инкапсуляции. Пример инкапсуляции.
32. Абстракция. Термины и определения, используемые в абстракции. Перечислить примеры абстракции.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Архангельский, А. Я. – Программирование в Delphi 6 / А. Я. Архангельский. – М.: ЗАО Издательство БИНОМ», 2002 г. – 1120 с. – Текст : непосредственный.

2. Свойства класса TApplication // Информатика и программирование: шаг за шагом : [сайт]. – URL: https://it.kgsu.ru/Delphi_6/dlph0018.html (дата обращения: 08.06.2025). – Текст : электронный.

3. Компонент ApplicationEvents // Информатика и программирование: шаг за шагом : [сайт]. – URL: https://it.kgsu.ru/Delphi_6/dlph0019.html (дата обращения: 08.06.2025). – Текст : электронный.

4. Vcl.Forms.TApplication.OnIdle // Embarcadero : [официальный сайт]. – URL: https://docwiki.embarcadero.com/Libraries/Athens/en/Vcl.Forms.TApplication.OnIdle (дата обращения: 08.06.2025). – Текст : электронный

5. AlexBin, ООП в картинках // Хабр : [сайт]. – URL: https://habr.com/ru/articles/463125/ (дата обращения: 03.01.2024). – Текст : электронный.

Учебное издание

Новиков Александр Игоревич

Алгоритмизация и программирование Delphi и Pascal

Часть 2

Учебно-методическое пособие

Редактор и корректор Д. А. Романова
Техн. редактор М. Д. Баранова Темплан 2024 г., поз. 5259

Подписано к печати 15.09.2025. Формат 60x84/16. Бумага тип № 1.
Печать офсетная. Печ.л. 6,9. Уч.-изд. л. 6,9.
Тираж 30 экз. Изд. № 74. Цена «С». Заказ №

Ризограф Высшей школы технологии и энергетики СПбГУПТД,
198095, Санкт-Петербург, ул. Ивана Черных, 4.