

Глава 25. Сплошная практика	606
25.1. Создание ScreenSaver.....	607
25.2. Компоненты в runtime.....	612
25.3. Тест на прочность.....	617
25.4. Сохранение и загрузка теста.....	628
25.5. Тестер.....	631



Глава 25. Сплошная практика

Я уже рассказал все необходимые основы. Теперь ты готов к самостоятельному изучению тонкостей программирования, а я только дам несколько практических примеров. В этой главе я покажу, как можно создавать собственные программы ScreenSaver и закрепим максимум из пройденного материала на полезных в реальной жизни примерах.

Те примеры, которые я писал на протяжении всей книги, очень хороши и удобны в образовательных целях. Но для полной готовности к реальной жизни надо ещё немного попрактиковаться, потому что описанные примеры были маленькие (в целях экономии места в книге) и теперь надо научиться всё сказанное собирать в единое целое.

Достаточно много дополнительного материала ты найдёшь на диске к этой книге. Я понимаю, что читать с монитора не так уж удобно, но я не хочу, чтобы эта книга была слишком дорогой. Моя Библия должна стать доступной каждому. При этом я постарался дать в ней максимум информации, при этом не просто заполнить компакт диск всякой ерундой, а наполнить его полезными вещами. Так что не ленись, и после прочтения книги загляни в директорию «Документация».



25.1. Создание ScreenSaver

Если ты до сих пор считаешь, что Delphi это язык, предназначенный для работы с базами данных, то в этой статье я окончательно опровергну твоё мнение. Здесь я покажу, что на Delphi можно написать даже хранитель экрана. Самое главное, что никаких особых усилий не понадобится.

С одной стороны, эта часть должна была идти в главе по графике, потому что мы будем много рисовать. Но я решил её вставить сюда, потому что используемые здесь графические функции мы уже изучили. А вот само программирование будет очень интересным.

Для того, чтобы Delphi мог создать хранитель экрана, ты должен сделать несколько установок для своей формы:

1. Перед объявлением типов вставить вот такую строку: `{SD SCRNSAVE Saver}`. Здесь *Saver* – имя нашего проекта, я его сохранил под именем *Saver.dpr*.
2. Свойство формы *BorderStyle* поменять на *bsNone*. Это означает, что у формы не должно быть никаких заголовков и обрамлений.
3. Все параметры в *Border Icons* установить в *False*.
4. Свойство формы *WindowState* поменять на *wsMaximized*, чтобы окно появлялось максимизированным на весь экран.
5. Создать событие *OnKeyPress* и вставить туда всего лишь одну процедуру - *Close()* – закрытие хранителя экрана по нажатию любой клавиши.
6. На событие *FormActivate* значения *Left* и *Top* нужно установить в ноль.

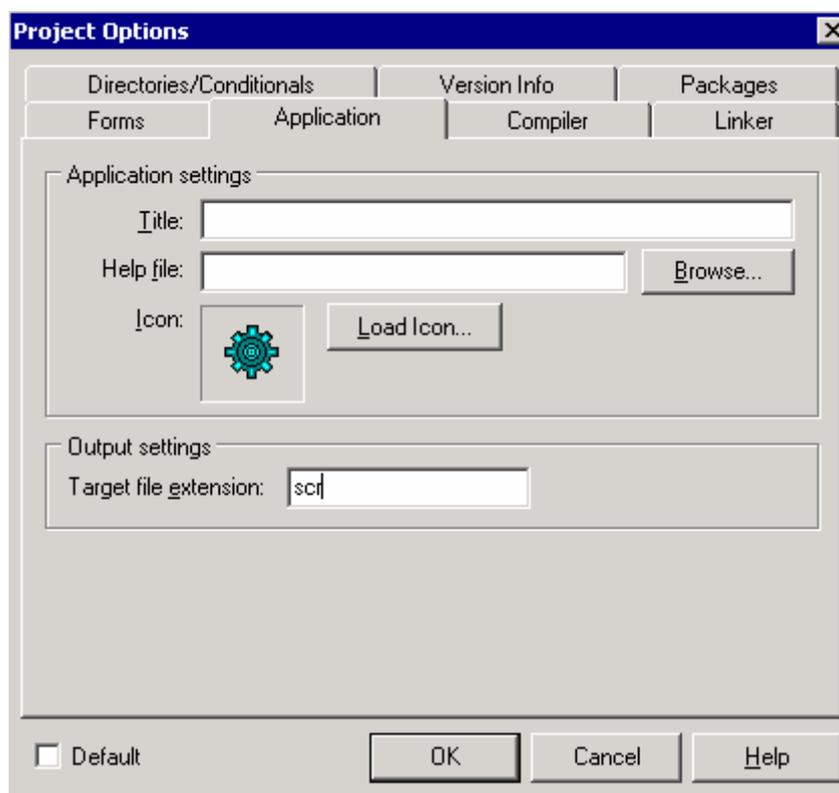


Рисунок 25.1.1 Свойства проекта

Теперь Delphi будек компилировать запускной файл, совместимый с хранителем экрана. Что такое ScreenSaver? Это та же программа, только с расширением *scr*. Так что

остаётся одна деталь – изменить расширение на *scr*. Ты можешь после компиляции программы переименовать файл в *.scr или возложить этот тяжёлый труд на Delphi. Для этого необходимо выбрать пункт Option из меню Project и на закладке *Application* в строке *Target file extension* написать scr (рисунок 25.1.1). В этом случае Delphi сам подставит это расширение.

Подготовительные работы закончены. Можно приступать к написанию хранителя экрана. Ты уже знаешь достаточно много, и сможешь сам написать что-нибудь интересное. А я здесь покажу простейший пример.

Для моего примера понадобится объявить три переменные в разделе **private**:

```
private
{ Private declarations }
BGbitmap:TBitmap;
DC : hDC;
BackgroundCanvas : TCanvas;
```

Затем посмотрим на мой обработчик события *OnCreate*:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  BGbitmap:=TBitmap.Create;//Инициализация

  // Выставляем размеры картинки как у экрана
  BGbitmap.Width := Screen.Width;
  BGbitmap.Height := Screen.Height;

  DC := GetDC (0);
  BackgroundCanvas := TCanvas.Create;
  BackgroundCanvas.Handle := DC;

  BGBitmap.Canvas.CopyRect(Rect (0, 0, Screen.Width, Screen.Height),
    BackgroundCanvas,
    Rect (0, 0, Screen.Width, Screen.Height));
  BackgroundCanvas.Free;
  randomize;
end;
```

Что здесь творится? В самом начале я инициализирую *BGbitmap* и устанавливаю ему размер как у экрана. Объект *Screen* – это объект, который создаётся автоматически при старте программы и содержит в себе информацию об экране. В свойствах *Width* и *Height* находятся ширина и высота экрана.

Потом в переменную *DC* заносится указатель на контекст вывода экрана. На первый взгляд это происходит не явно, но всё очень просто. Функция *GetDC* возвращает указатель на контекст указанного в качестве параметра устройства или формы. Если указать 0, то *GetDC* вернёт указатель на контекст воспроизведения экрана. Если ты захочешь получить контекст воспроизведения твоей формы, то ты должен написать *GetDC(Handle)*. В качестве параметра выступает *Handle* (указатель) окна. Но ты в таком виде не будешь никогда использовать *GetDC*, потому что у тебя уже есть контекст воспроизведения формы - это *Canvas*, а указатель на него - *Canvas.Handle*.

Дальше я создаю *BackgroundCanvas* - это *TCanvas*, который будет указывать на экран. Я делаю это только для удобства. Я мог бы использовать для рисования *DC*, но всё

же *TCanvas* более понятен и мы с ним достаточно подробно разобрались в главе посвящённой графике.

С помощью следующей строки, я сохраняю копию экрана:

```
BGBitmap.Canvas.CopyRect(Rect (0, 0, Screen.Width, Screen.Height),  
BackgroundCanvas,  
Rect (0, 0, Screen.Width, Screen.Height));
```

Затем я уничтожаю указатель на контекст экрана *BackgroundCanvas.Free*, потому что больше он нам не понадобится.

Самая последняя функция - *randomize* инициализирует таблицу случайных чисел. Если ты этого не сделаешь, то после запроса у системы случайного числа, тебе вернут значение из текущей таблицы, которая не всегда удачна. Тебе не надо будет видеть таблицу случайных чисел, она прекрасно будет работать без твоих глаз.

В обработчике события *OnDestroy* я уничтожаю картинку:

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

Последняя функция - это обработчик таймера, который я поставил на форму:

```
procedure TForm1.Timer1Timer(Sender: TObject);  
const  
DrawColors: array[0..7] of TColor =(clRed, clBlue, clYellow, clGreen,  
clAqua, clFuchsia, clMaroon, clSilver);  
begin  
BGBitmap.Canvas.Pen.Color:=DrawColors[random(7)];  
BGBitmap.Canvas.MoveTo(random(Screen.Width),random(Screen.Height));  
BGBitmap.Canvas.LineTo(random(Screen.Width),random(Screen.Height));  
Canvas.Draw(0,0,BGBitmap);  
end;
```

В разделе констант я объявляю массив *DrawColors*, который хранит восемь цветов. Объявление происходит следующим образом:

*Имя_Массива : array [Индекс_первого_значения .. Индекс_последнего_значения] of
Тип_значения_Массива = (Перечисление_значений)*

В "перечислении значений" должно быть описано "Индекс последнего значения" - "Индекс первого значения"+1 параметров. В моём случае это 7-0+1=8 значений цвета.

Дальше меняю цвет у контекста рисования формы *BGBitmap.Canvas.Pen.Color* случайным цветом из массива *DrawColors*. Функция *random* возвращает число из таблицы случайных чисел, при этом, это число будет больше нуля и меньше значения переданного в качестве параметра. Это значит, что *random(7)* вернёт случайное число от 0 до 7. С помощью *BGBitmap.Canvas.MoveTo* я перемещаюсь в случайную точку внутри *BGBitmap* и с помощью *BGBitmap.Canvas.LineTo* рисую из этой точки в новую точку линию. *Canvas.Draw(0,0,BGBitmap)* выводит моё произведение на экран.

Попробуй запустить пример, и ты увидишь, как экран засыпается линиями со случайными координатами. Но если ты протестируешь пример, то заметишь несколько недостатков:

1. Если скопировать хранитель экрана в системную директорию (windows или winnt) и попытаться установить этот хранитель, то будут заметны блики.

2. При нажатии на кнопку «Настроить» запускается хранитель, а не окно настройки.

Как же нашему хранителю экрана узнать, что надо отображать окно настройки, а не запускаться самому? Очень просто. При старте хранителя экрана нам в командной строке передаются параметры. Они могут быть следующими:

/s – надо запустить хранитель экрана.

/p – хранитель экрана должен отображаться в окне свойств экрана на рисунке монитора (рисунок 25.1.2).

/c:xxxxx – нужно отобразить окно настроек. Здесь после двоеточия, вместо xxxx стоит число.

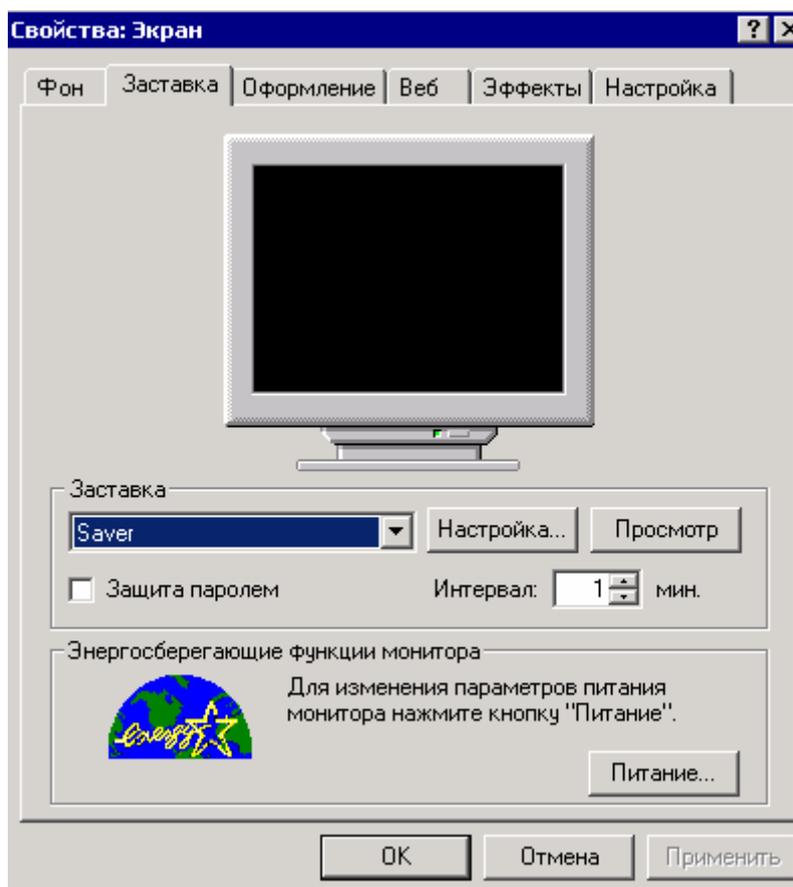


Рисунок 25.1.2 Свойства экрана

Количество параметров переданных программе можно узнать, вызвав функцию *ParamCount*. Эта функция вернёт только количество параметров. Сами параметры можно получить с помощью функции *ParamStr*. Этой функции нужно только передать номер параметра, который мы хотим получить. Чтобы перебрать все переданные программе значения можно использовать следующий код:

```
var  
i:Integer;  
begin
```

```
for i:=0 to ParamCount-1 do
  Переменная:=ParamStr(i);
end;
```

Теперь забудем на минутку про параметры и для начала избавимся от бликов. Для этого по событию *OnCreate* в самом начале присваиваем ширине и высоте окна значения 0, чтобы окно было минимальным. В этом случае окно не будет мерцать, потому что его просто нет на экране из-за нулевой ширины и высоты.

```
Width:=0;
Height:=0;
```

А вот теперь вернёмся к нашим параметрам. По событию *OnShow* для главной формы пишем следующий код:

```
procedure TSaverForm.FormShow(Sender: TObject);
begin
  if ParamCount>0 then
  begin
    if ParamStr(1)='/p' then
    begin
      Close;
      exit;
    end;
    if ParamStr(1)[2]='c' then
    begin
      OptionsForm.ShowModal;
      Close;
      exit;
    end;
  end;

  Timer1.Enabled:=true;
  Width:=Screen.Width;
  Height:=Screen.Height;
end;
```

В самом начале я проверяю, если количество параметров больше нуля, то нужно будет проверить, что нам передали. Если параметр равен '/p' то мы просто будем закрывать хранитель экрана и ничего отображать не будем. Если параметр равен '/с:xxxx', то будем отображать окно настроек. Но здесь с проверкой небольшое затруднение, ведь мы не знаем, какое число будет вместо xxxx. Поэтому я просто проверяю второй символ параметра и если он равен букве «с», то значит это «/с».

Окно свойств я сделал простейшим (смотри рисунок 25.1.3), в нём всего лишь запрашивается пароль для хранителя экрана. Саму реализацию работы с паролем я опустил, потому что здесь всё просто. Тебе нужно сохранять этот пароль в реестре, а при попытке выхода из программы запрашивать ввод пароля. Если пользователь ввёл правильный пароль, то хранитель экрана можно закрывать. Попробуй всю эту логику написать сам. Если что-то не получится, то на диске сможешь найти мой пример, в котором реализовано всё необходимое.

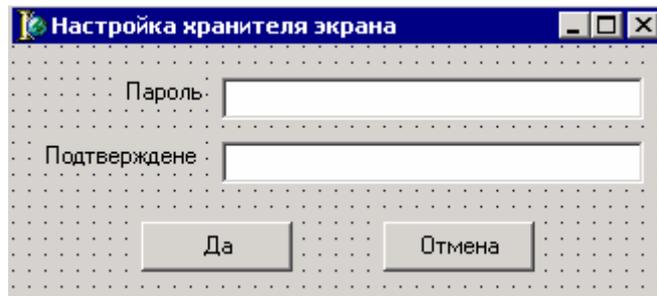


Рисунок 25.1.3 Окно настроек хранителя экрана

Теперь вернёмся к нашему обработчику события *OnShow*. После проверки всех параметров, я делаю таймер *Timer1* активным. Если у тебя на форме стоит активный таймер (свойство *Enabled* равно *true*), то сделай его неактивным.

После активизации таймера я выставляю ширину и высоту окна в значения ширины и высоты всего экрана, чтобы моё окно покрыло весь экран.

Остальной код практически не изменился, за исключением того, что ты должен добавить проверку пароля. Я могу тебе ещё посоветовать добавить возможность управления частотой таймера в окне настроек, но это уже по желанию.

 На компакт диске, в директории *\Примеры\Глава 25\ScreenSaver* ты можешь увидеть пример этой программы.

25.2. Компоненты в runtime

Здесь я опишу маленький пример, который ответит сразу на два поставленных мне вопроса: как создавать компоненты во время выполнения программы и как ими управлять. Что такое runtime? Твоя прога может находиться в двух состояниях - *designtime* (время создания проекта) и *runtime* (время выполнения проекта). Мы сегодня будем создавать компоненты не рисованием на форме, а чистым кодом уже во время выполнения программы.

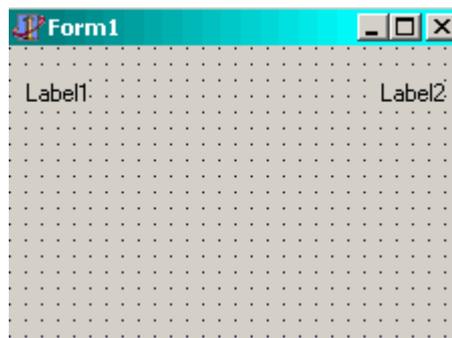


Рис 25.2.1. Форма будущей программы

Создай новый проект и брось на него два компонента *TLabel* рисунок 25.2.1. Всё остальное будем делать ручками. Для начала, в разделе **private** объявим переменную *CompList* типа *TList*. *TList* - это "объект-контейнер", который может хранить в себе кучу других. Точнее сказать, он хранит только ссылки, но это не главное. Главное - *TList* позволяет хорошо управлять хранящимися в нём объектами.

На событие *OnCreate* напиши:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  CompList:=TList.Create;
end;
```

Здесь мы инициализируем нашу переменную *CompList* с помощью объекта *TList*. Во время инициализации выделяется память под нашу переменную. Сразу же на событие *OnDestroy* пишем:

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

Здесь мы освобождаем выделенную память для переменной *CompList*.

Теперь создадим обработчик события нажатия мышкой *OnMouseDown*. По нажатию кнопкой, мы будем создавать на форме новый компонент *TPanel*. Давай в нём напишем следующий код:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  TempPanel:TPanel;//Объявляю переменную для панели
begin
  //Создаю панель. В скобках у Create указан будущий владелец
  TempPanel:=TPanel.Create(Form1);

  TempPanel.Left:=X; //Устанавливаю левую и правую координату
  TempPanel.Top:=Y; //в X и Y позицию, где нажата кнопка мыши

  TempPanel.Width:=20; //Устанавливаю ширину
  TempPanel.Height:=20; //Устанавливаю высоту

  //Далее устанавливаю обработчик нажатия на эту панель
  TempPanel.OnMouseDown:=PanelMouseDown;

  //Добавляю панель в контейнер CompList (CompList.Add)
  //и сохраняю результат в TempPanel.Tag
  TempPanel.Tag:=CompList.Add(TempPanel);

  Form1.InsertControl(TempPanel); //Вставляю панель на форму
end;
```

Для начала вспомним, что это за свойство *Tag* у компонента *TPanel*. Это просто целое значение, которое ты можешь использовать по своему усмотрению. Именно этим свойством мы и будем часто пользоваться во время программирования нашего примера.

Теперь разберём написанный код. В разделе **var** я объявил одну переменную *TempPanel* типа *TPanel*. Это временная переменная, в которой будет инициализироваться новая панель. В первой же строчке кода обработчика я инициализирую эту переменную, как панель. В качестве параметра методу *Create* я должен передавать имя объекта,

который будет являться родителем создаваемого компонента. Я передаю нашу главную форму, потому что компонент будет размещаться именно на нём.

Следующим этапом, я устанавливаю левую и правую позицию панели в координаты, где мы щёлкнули мышкой (X и Y, которые нам переданы в обработчике, указывают на точку, в которой была нажата кнопка мышки). Далее, я устанавливаю ширину и высоту панели. Я решил занести туда значение 20 (просто так захотелось).

Теперь об обработчике события *TempPanel.OnMouseDown*. Я туда засунул имя функции *PanelMouseDown*. Но такой функции нет среди стандартных функций и среди моего проекта. Поэтому мы должны её создать сами. Как это сделать эффективно? Вот тебе мой совет:

1. Мы создаём обработчик для *TPanel*, поэтому временно поставь один экземпляр панели на форму в произвольное место.

2. Создай для него обработчик на *OnMouseDown* и переименуй его в *PanelMouseDown*.

3. Напиши нужный текст (я его покажу ниже) и можно удалять временно созданный на форме экземпляр *TPanel*.

Таким образом, ты можешь быть уверен, что ошибок не будет, потому что Delphi сама пропишет функцию *PanelMouseDown* где надо и укажет все необходимые параметры.

Если захочешь объявлять эту функцию вручную, то напиши в разделе **private**:

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

Объявлять можно и до **private**, там где объявляет Delphi обработчики событий. А ниже опиши саму функцию

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

Обязательно нужно следить, чтобы количество и тип параметров точно совпадали с необходимыми. У каждого обработчика свои параметры и при объявлении процедуры вручную, нужно относиться к этому вопросу очень внимательно. Именно поэтому я советую тебе первый способ, с временной панелью, когда Delphi сам создаст обработчик для одной панели, а ты будешь использовать его для других.

Всё, панель готова и её надо сохранить в нашем контейнере *CompList*. Для этого нужно выполнить метод *Add* нашего контейнера, в качестве параметра передать ему нашу панель:

```
CompList.Add(TempPanel)
```

Этот метод добавит панель в контейнер и вернёт нам индекс компонента в контейнере. Этот индекс я сохраняю в свойстве *Tag* нашей панели *TempPanel*. Это свойство абсолютно не влияет на сам компонент, а мне этот индекс пригодится.

Теперь давай посмотрим на функцию *PanelMouseDown*, которая должна быть такой:

```
procedure TForm1.PanelMouseDown(Sender: TObject; Button: TMouseButton;
```

```

Shift: TShiftState; X, Y: Integer);
begin
Label1.Caption:=IntToStr(TPanel(CompList.Items[TPanel(Sender).Tag]).Left);
Label2.Caption:=IntToStr(TPanel(Sender).Left);
end;

```

Здесь две строки. Обе строки выполняют одно и тоже, но по-разному. Обе строки записывают в свой TLabel левую позицию панели, по которой ты щёлкнул.

Первая строка, чтобы получить левую позицию панели использует CompList, а вторая работает с панелью напрямую. Рассмотрим сначала вторую строку. В ней основным является выражение *TPanel(Sender).Left*. *Sender* - передаётся нам процедурой обработчиком *PanelMouseDown*. В нём записан указатель на объект, который сгенерировал событие *OnMouseDown*. В нашем случае это будет указатель на панель, по которой ты щёлкнул. Так как мы точно уверены, что это панель, то мы так и показываем *TPanel(Sender)*. Этим мы приводим *Sender* к *TPanel* и теперь ты можешь использовать все свойства и методы панели, для примера нам достаточно свойства *Left*. Если бы мы знали точное имя панели, то этого писать не пришлось бы. Но это невозможно, потому что все создаваемые в Runtime панели (а их можно создать любое количество) у нас используют один обработчик нажатия мышкой, и мы не знаем, по какой именно панели был произведён щелчок. Получив значение левой позиции, мы переводим целое значение левой позиции в строку с помощью *IntToStr*.

Первая строка очень похожа на вторую, только внутри *TPanel()* мы используем не *Sender*, а *CompList.Items[TPanel(Sender).Tag]*, т.е. значение из контейнера. Чтобы получить первое значение из контейнера, нужно написать *CompList.Items[0]*, для второго *CompList.Items[1]*, для третьего *CompList.Items[2]* и т.д. Но по какой именно панели произведён щелчок? Чтобы это узнать я пишу *TPanel(Sender).Tag*, то есть получаю свойство *Tag* (там хранятся индекс панели) панели сгенерировавшей событие. Далее, всё происходит так же.

Запусти пример и пощёлкай по форме. По каждому щелчку будут создаваться панели. Потом попробуй пощёлкать по самим панелям. На двух TLabel будут появляться значения левой позиции панель, по которым ты щёлкал.

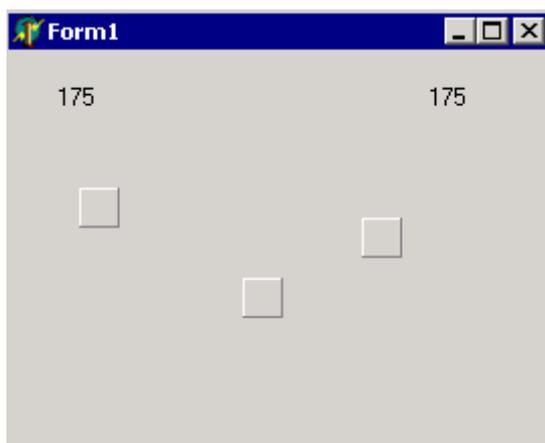


Рисунок 25.2.2 Пример работы программы.

Теперь я хочу тебе показать ещё несколько интересных свойств и методов, которые есть у контейнера *TList*:

Count – в этом свойстве храниться количество элементов в контейнере.

Items – здесь хранятся ссылки на элементы контейнера. Для доступа к ссылкам нужно написать *Items[Индекс элемента]*.

Clear – очистить список.

Delete – удалить элемент из списка. В качестве единственного параметра нужно указать индекс удаляемого элемента.

Exchange – поменять в контейнере местами два элемента. Здесь два параметра – индексы меняемых местами компонентов.

First – получить указатель на первый элемент списка. Это то же самое, что и записать *Items[0]*.

IndexOf – получить индекс указанного в качестве параметра объекта. Допустим, что ты знаешь объект (*TPanel*) и хочешь узнать, под каким индексом он расположен в контейнере. В этом случае ты можешь написать следующий код: *CompList.IndexOf(Panell)*. Если такой панели не найдено в списке, то тебе будет возвращено значение -1 , иначе правильный индекс указанной панели.

Insert – вставить новый элемент. У этого метода два параметра – индекс, под которым надо вставить элемент и сам элемент.

Last – получить последний элемент списка. Это то же самое, что использовать свойство *Items[Count - 1]*.

Move – переместить элемент в новое место. У метода два параметра – индекс элемента, который надо переместить и индекс, который должен получить элемент.

Pack – при удалении элементов из контейнера, они просто помечаются, как нулевые. Выполняя этот метод, все нулевые элементы уничтожаются, и занятая ими память освобождается.

Remove – удалить элемент. В качестве параметра нужно указывать элемент, который надо удалить, например *CompList.Remove(Panell)*.

Давай добавим в наш пример возможность удаления панели с формы и из контейнера. Это будет происходить по нажатию правой кнопки мышки по компоненту. Добавь с конец процедуры *PanelMouseDown* следующий код:

```
if Button=mbRight then
begin
  index:=TPanel(Sender).Tag;
  TPanel(CompList.Items[index]).Free;
  CompList.Delete(index);

  for i:=index to CompList.Count -1 do
    TPanel(CompList.Items[i]).Tag:=TPanel(CompList.Items[i]).Tag-1;
end;
```

В разделе **var** этой процедуры нужно объявить две переменные *index* и *i*. Обе они будут числами целого типа.

Теперь разберём код. Сначала я проверяю, если нажата правая кнопка мыши, то нужно удалить компонент, по которому щёлкнули. Для этого, я сначала сохраняю индекс компонента *TPanel(Sender).Tag* в переменной *index*. Это необходимо, потому что после уничтожения компонента *TPanel(Sender)* не будет существовать, и я не смогу получить доступ к его свойству *Tag*.

Следующим этапом происходит удаление. Сначала я уничтожаю сам компонент - *TPanel(CompList.Items[index]).Free*, а после это уничтожаю ссылку на него в контейнере - *CompList.Delete(index)*. Вроде бы всё удалили, но программа после этого будет работать неправильно. Допустим, что у нас в контейнере было 5 элементов, и мы удалили 3-й. По

идее, в контейнере должны остаться элементы с индексами 1, 2, 4, 5, а 3-й должен отсутствовать. Но реально индексы перестроятся, и мы увидим индексы 1, 2, 3, 4. Если мы попытаемся оставить всё так, как есть, то программа будет нестабильна. Если ты щёлкнешь, по последней созданной панели, то программа обратится к элементу в контейнере под номером 5, потому что в свойстве *Tag* панели находится цифра 5. Но такого элемента не существует и произойдёт ошибка. Поэтому нам надо подкорректировать свойства *Tag* у всех панелей начиная с удаляемой. Для этого я запускаю цикл от индекса удаляемой панели до последнего элемента списка и уменьшаю их свойство *Tag*:

```
for i:=index to CompList.Count -1 do  
  TPanel(CompList.Items[i]).Tag:=TPanel(CompList.Items[i]).Tag-1;
```

Вот теперь у меня в контейнере будут элементы с индексами от 1 до 4 и у всех панелей на форме в свойстве *Tag* будут правильные значения.

 *На компакт диске, в директории \Примеры\Глава 25\Runtime ты можешь увидеть пример этой программы.*

25.3. Тест на прочность.

Ко мне очень часто приходят вопросы, хоть как-то связанные с написанием программы теста. Видимо преподаватели в институтах начинают любить компьютеры, и используют их для тестирования знаний учеников. Но так как в наших школах есть проблемы с программным обеспечением, то преподаватели дают задания своим ученикам-программистам написать какой-нибудь определённый тест.

В этой книге я решил подробно описать процесс написания программы теста, потому что в такой программе будет множество интересных приёмов программирования и в для обучения такая программа будет просто идеальной.

Итак, нам придётся написать две программы:

1. Редактор тестов. В ней будут создаваться тесты, заполняться вопросы, на которые надо будет отвечать и варианты правильных ответов.
2. Программа тестирования. Это оболочка, которая будет загружать созданные тесты, и в ней пользователь должен будет отвечать на появляющиеся вопросы.

Итак, начнём мы с редактора, потому что сначала нужно научиться создавать тесты, а потом уже будем их отображать и работать с ними. Для начала я создал главную форму, как показано на рисунке 25.3.1. Здесь у меня главное окно, в котором есть главное меню программы, панель кнопок быстрого вызова команд *ToolBar* и строка состояния, которая будет отображать подсказки. Попробуй создать что-то подобное.

На форме у меня созданы следующие кнопки (и соответствующий пункты меню):

1. Создать;
2. Открыть;
3. Сохранить;
4. Печать;
5. Настройки программы;
6. Помощь;
7. О программе;
8. Выход.

У каждой кнопки в свойстве *Hint* указана соответствующая ей подсказка, которая должна появляться в строке состояния при наведении на кнопку. Чтобы эта подсказка появлялась и рядом с кнопкой, установи свойство *ShowHint* у нашей главной формы в значение *true*.

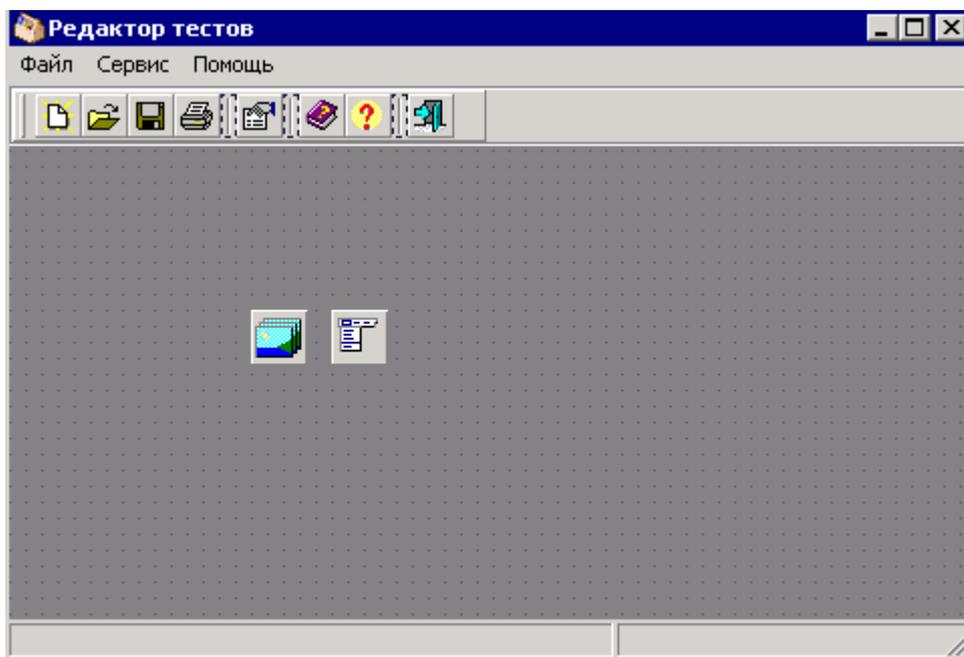


Рисунок 25.3.1. Главная форма редактора теста.

Первым делом, давай сразу же сделаем возможность отображения подсказок *Hint* в строке состояния. Для этого в разделе **private** опиши новую процедуру:

```
private
{ Private declarations }
procedure ShowHint(Sender: TObject);
```

Теперь нажми **Ctrl+Shift+C**, чтобы создать заготовку для этой процедуры. В ней нужно написать следующее:

```
procedure TTestEditorForm.ShowHint(Sender: TObject);
begin
  StatusBar.Panels.Items[0].Text := Application.Hint;
end;
```

Здесь я отображаю текст текущей подсказки, который находится в свойстве *Hint* объекта *Application* на нулевой панели строки состояния. У меня строка состояния состоит из двух панелей.

Теперь сделай нашу главную форму многодокументной. Для этого в свойстве *FormStyle* нужно установить значение *fsMDIForm*.

Можешь сразу же создать окно «О программе». Я это не буду описывать, потому что тут фантазия у каждого работает по-разному, а моё окно ты сможешь увидеть на диске вместе с исходником. Можешь ещё сразу создать обработчик события *OnClick* для кнопки

выхода и написать там вызов метода *Close*, чтобы по нажатию этой кнопки наша программа закрывалась. Потом, эту же процедуру обработчик нужно назначит пункту меню «Выход».

На этом первые приготовления закончены. Теперь двинемся дальше. Создай новую форму (я её назвал *NewTestForm*), которая будет отображаться по нажатию кнопки «Создать». В этом окне нужно расположить строку ввода имени теста и выпадающий список для выбора типа теста. Я разместил все компоненты так, как показано на рисунке 25.3.2

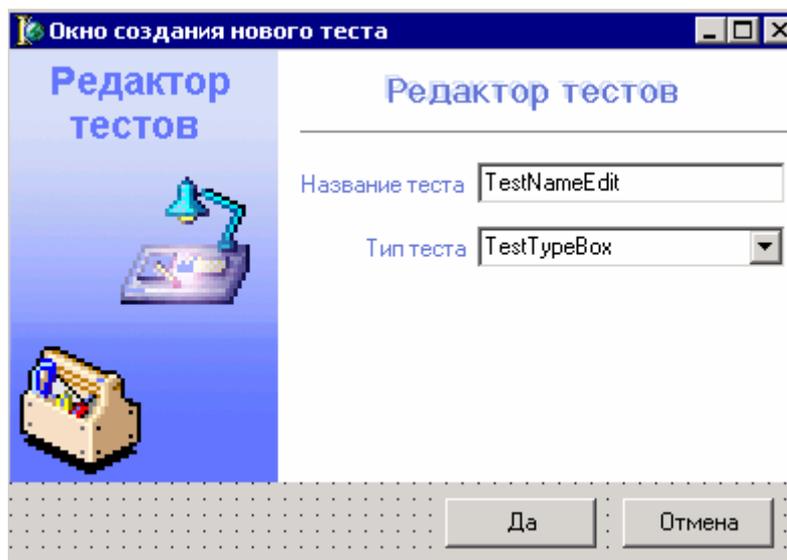


Рисунок 25.3.2. Окно создания нового теста.

У выпадающего списка я изменил свойство *Style* на *csDropDownList* для того, чтобы пользователь мог только выбирать уже имеющееся значение в списке и не мог вводить новое. В свойстве *Items* (список элементов) у меня будет только одна строка – «Вопрос - варианты ответа». В принципе, можно было бы не делать этот выпадающий список, раз там только один элемент, но я его поставил, чтобы ты потом мог расширить возможности программы.

Для кнопок «Да» и «Нет» я установил только свойство *ModalResult* в значения *mrOk* и *mrCancel* соответственно.

У самой формы я изменил свойства:

Position на *poMainFormCenter*, чтобы окно показывалось по центру главного окна.

BorderStyle на *bsSingle*, чтобы пользователь не мог изменять размеры окна.

Эти свойства я буду менять у всех форм, которые будут отображаться модально. Кнопки «Да» и «Нет» у таких окон тоже всегда будут иметь указанные выше значения свойства *ModalResult*, поэтому я больше не буду останавливаться на этих вещах. Просто помни, это.

Теперь возвращаемся в главную форму и по событию *OnClick* для кнопки создать пишем код отображения окна *NewTestForm*. Это окно нужно отображать как модальное.

На этом остановимся и сделаем промежуточную паузу.

 На компакт диске, в директории \Примеры\Глава 25\Test1\Редактор ты можешь увидеть исходник уже написанного примера.

Теперь создадим окно создания нового теста вопросов ответов. Создай новую форму и установи у неё следующие свойства:

Caption – «Тест "Вопрос - варианты ответов"»

FormStyle – *fsMDIChild*, это у нас будет дочернее окно.
Name – *QuestionResultForm*.

По событию *OnClose* пишем следующий код:

```
procedure TQuestionResultForm.FormClose(Sender: TObject;  
  var Action: TCloseAction);  
begin  
  Action:=caFree;  
  QuestionResultForm:=nil;  
end;
```

Здесь, в первой строке я устанавливаю переменной *Action* (эту переменную мы получаем в качестве параметра) значение *caFree*, чтобы окно могло закрыться. Дочерние окна по умолчанию не закрываются, а сворачиваются. Во второй строке я обнуляю переменную *QuestionResultForm*, которая указывает на объект окна.

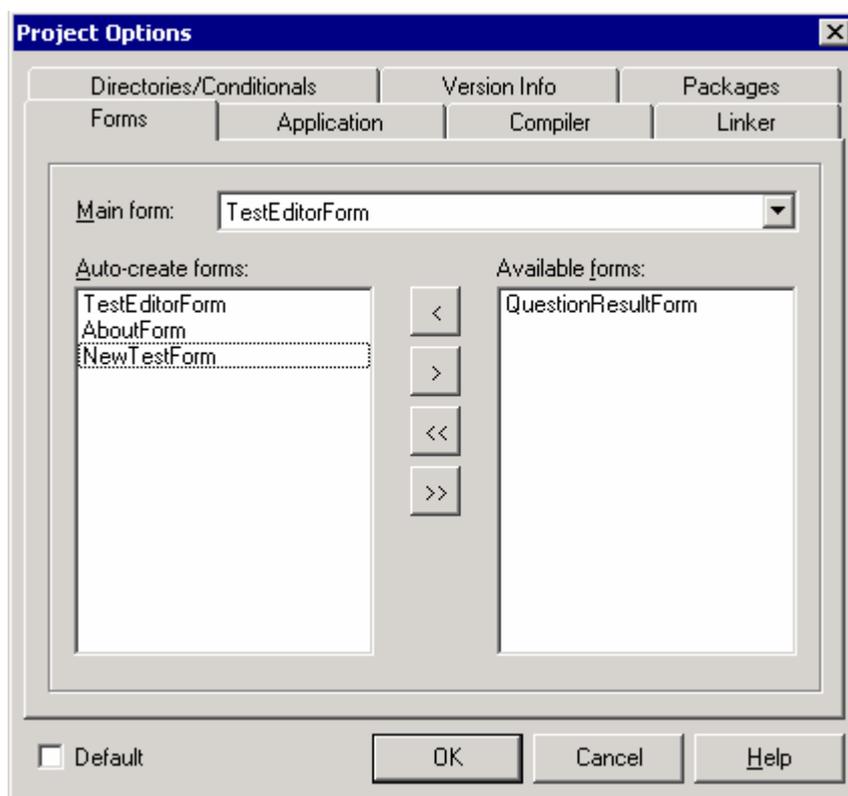


Рисунок 25.3.3 Окно свойств проекта

Дочерние окна при старте автоматически становятся видимыми и отображаются в окне главного окна. Нам это не надо, потому что это окно должно появляться только после того, как пользователь нажмёт кнопку создания нового теста и подтвердит его создание. Для того, чтобы отключить форму от автоматического создания, мы должны войти в свойства проекта (Project->Options) и в появившемся окне переместить имя формы *QuestionResultForm* из списка *Auto-create forms* в список *Available forms* (рисунок 25.3.4).

Вот теперь перейдём к дизайну формы создания теста. Мою форму ты можешь увидеть на рисунке 25.3.4. В центре окна у меня находится компонент *TreeView*

растянутый по левому краю и *ListView* растянутый по всему оставшемуся контуру (сlClient). Сверху окна расположена панель *ToolBar* со следующими кнопками:

1. Создание вопроса.
2. Редактирование.
3. Удаление.
4. Выход.

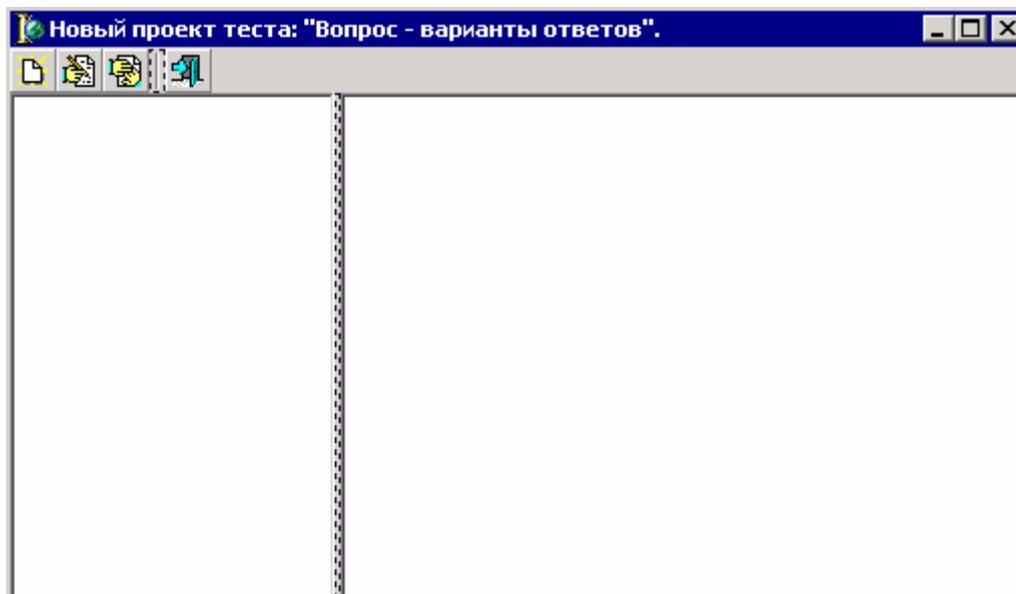


Рисунок 25.3.4 Форма окна создания теста

Выдели компонент *ListView* и установи у него следующие свойства:

1. *GridLines* – *true*.
2. *ViewStyle* – *vsReport*.
3. *Name* - *ResultView*.

После этого дважды щёлкни по свойству *Columns* и в появившемся окне создай две колонки с именами «Верно» и «Вариант ответа». У второй колонки установи свойство *AutoSize* в значение *true*.

Теперь объявим в разделе **type** следующую структуру:

```
PQuestion=^TQuestion;  
TQuestion=record  
  Name: String[255];  
  ResultCount: Integer;  
  ResultText: array[0..10] of String[255];  
  ResultValue: array[0..10] of boolean;  
end;
```

Я объявил структуру *TQuestion* со следующими полями:

Name – здесь будет храниться вопрос.

ResultCount - количество вариантов ответов.

ResultText – массив из строк для десяти вариантов ответов.

ResultValue – массив из булевых переменных, указывающих, какие ответы верные.

Тут же объявлена переменная *PQuestion*, которая является указателем на структуру *TQuestion*.

Теперь в разделе **public** объявим следующие переменные:

```
public
{ Public declarations }
ProjectName:String[255];
QuestionList:TList;
```

В переменной *ProjectName* будем хранить имя проекта. Список *QuestionList* будет использоваться для хранения структур типа *PQuestion*. Этот список должен инициализироваться по событию *OnShow* и уничтожаться по событию *OnClose*. Как это делать ты уже должен знать.

Теперь создаём форму для редактирования элементов, которую мы будем отображать на экране по нажатию кнопки «Создать вопрос» и «Редактировать вопрос». Мою форму ты можешь увидеть на рисунке 25.3.5.

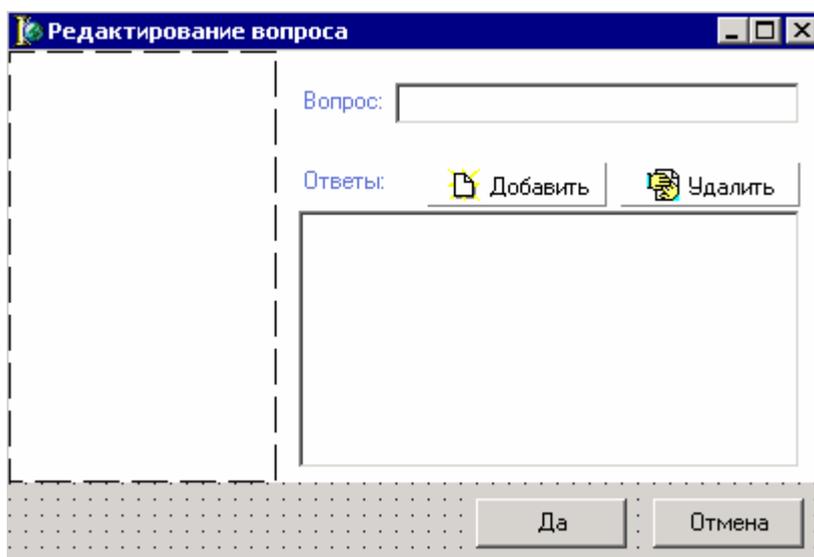


Рисунок 25.3.5 Форма окна ввода вопросов

Здесь находится строка *TEdit*, для ввода текста вопроса и *TCheckBox* для ввода вариантов ответа. По нажатию кнопки «Добавить» я буду показывать окно ввода нового варианта ответа:

```
procedure TEditQuestionForm.NewResultButtonClick(Sender: TObject);
var
  Str:String;
begin
  Str:="";
  if InputQuery('Новый ответ', 'Введите текст ответа:', Str) then
    ResultListBox.Items.Add(Str);
end;
```

Здесь я объявил одну строковую переменную. В первой строке кода ей присваивается пустая строка. Затем я отображаю на экране стандартный диалог ввода текстовой строки с помощью функции *InputQuery*. На рисунке 25.3.6 ты можешь видеть пример такого окна. У этой функции три параметра:

1. Текст, который будет отображаться в заголовке окна.
2. Текст, который будет отображаться в окне, рядом со строкой ввода.
3. Строковая переменная, через которую мы можем передать значение по умолчанию и получить результат ввода.

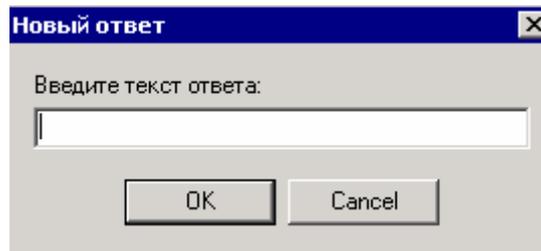


Рисунок 25.3.6 Форма окна создания теста

Если функция возвращает *true*, то пользователь после ввода нажал кнопку *OK*, и в этом случае я добавляю введённый текст в список ответов *ResultListBox*.

По нажатию кнопки «Удалить» я пишу следующий код:

```
procedure TEditQuestionForm.SpeedButton1Click(Sender: TObject);
begin
  if ResultListBox.ItemIndex<>-1 then
    ResultListBox.Items.Delete(ResultListBox.ItemIndex);
end;
```

В первой строчке кода я проверяю свойство *ItemIndex*, которое указывает на выделенный элемент в списке. Если оно не равно -1 , значит, в списке есть выделенный элемент, и я его должен удалить. Для этого я выполняю *ResultListBox.Items.Delete* указывая в качестве параметра выделенную строку.

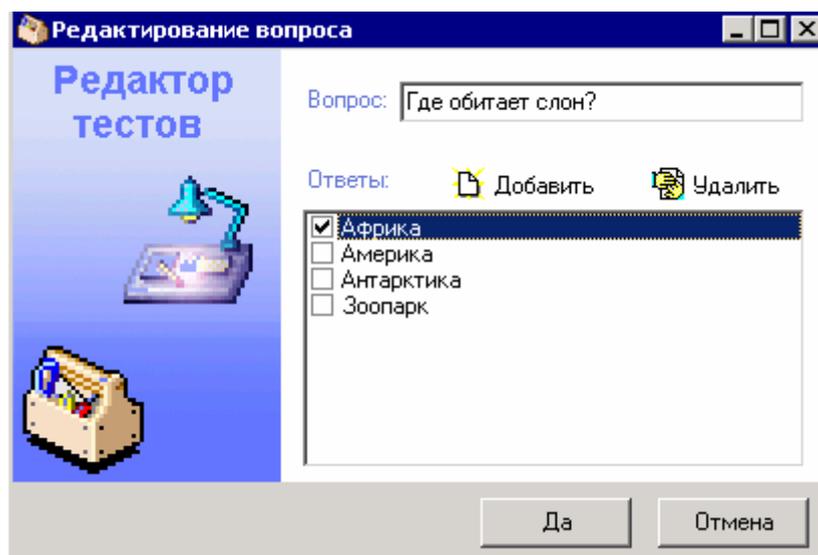


Рисунок 25.3.7 Окно редактирования вопроса в действии.

Теперь вернёмся к нашему окну *QuestionResultForm*. Здесь создадим обработчик события *OnClick* для кнопки создания нового вопроса:

```

procedure TQuestionResultForm.NewButtonClick(Sender: TObject);
var
  NewQuest:PQuestion;
  i:Integer;
begin
  //Очищаю содержимое окна EditQuestionForm
  EditQuestionForm.ResultListBox.Items.Clear;
  EditQuestionForm.QuestionEdit.Text:="";

  //Отображаю окно на экране
  EditQuestionForm.ShowModal;
  if EditQuestionForm.ModalResult<>mrOK then exit;

  //Создаю в памяти новую структуру
  NewQuest:=New(PQuestion);
  NewQuest.Name:=EditQuestionForm.QuestionEdit.Text;
  NewQuest.ResultCount:=EditQuestionForm.ResultListBox.Items.Count;

  //Добавляю в структуру варианты ответов
  for i:= 0 to NewQuest.ResultCount-1 do
  begin
    NewQuest.ResultText[i]:=EditQuestionForm.ResultListBox.Items.Strings[i];
    NewQuest.ResultValue[i]:=EditQuestionForm.ResultListBox.Checked[i];
  end;
  QuestionList.Add(NewQuest);

  //Добавляю новый элемент в дерево вопросов
  with QuestionTreeView.Items.Add(nil, NewQuest.Name) do
  begin
    ImageIndex:=0;
    Data:=NewQuest;
  end;
end;

```

В самом начале я очищаю элементы управления окна *EditQuestionForm*. Потом я отображаю это окно, и если пользователь ввёл название вопроса и нажал *OK*, то нужно обработать введённую информацию. Для начала выделяется память под переменную *NewQuest*. Эта переменная объявлена как *PQuestion*, а это указатель на структуру *TQuestion*. Как ты знаешь, любые указатели создаются пустыми и чтобы они на что-то указывали, им надо выделять память. Я выделяю память с помощью функции *New*. Этой функции нужно передать в качестве параметра подо что нужно выделять память. Я указываю наш указатель *PQuestion*, по которому функция определит, сколько памяти надо выделить. Результат выполнения функции – указатель на выделенную память, который я сохраняю в переменной *NewQuest*.

Если нужно уничтожить выделенную память, то мы должны вызвать процедуру *Dispose* и передать ей переменную, которую нужно уничтожить, например *Dispose(NewQuest)*. Но мне не надо уничтожать эту переменную, потому что она потом будет использоваться и я её добавляю в список *QuestionList* типа *TList*.

После того, как я выделил память для структуры *NewQuest*, я заполняю её поля в зависимости от введённой пользователем информации. Как только всё заполнено, я добавляю структуру в список:

```

QuestionList.Add(NewQuest);

```

После этого происходит самое интересное. Я должен создать новый элемент в дереве вопросов. Для этого выполняется следующий код:

```
with QuestionTreeView.Items.Add(nil, NewQuest.Name) do
begin
  ImageIndex:=0;
  Data:=NewQuest;
end;
```

Давай разберём этот код по частям. В первой строке я добавляю в дерево новый элемент с помощью вызова *QuestionTreeView.Items.Add*. В качестве параметров методу *Add* нужно передать указатель на родительский элемент в дереве и текст элемента. В качестве родительского элемента я передаю *nil* потому что я буду создавать дерево без вложенных элементов. В качестве текста элемента я передаю текст вопроса.

Выполненный метод *Add* возвращает указатель на созданный элемент. И тут у меня стоит оператор **with**, который заставляет выполнять следующие действия с указанным объектом. Получается, что следующие действия между **begin** и **end** будут выполняться с созданным элементом дерева вопросов. А у меня тут выполняется два действия:

ImageIndex:=0 – индексу иконки присваивается значение 0. Я бросил на форму список картинок *ImageList*, загрузил туда несколько картинок и указал этот список в свойстве *Images* нашего дерева. В этом коде я назначаю элементу первую картинку из созданного списка.

Data:=NewQuest – Свойство *Data* элемента дерева – это такое же свойство, как *Tag* у всех компонентов. Оно так же не влияет на работу компонента и его элементов и может использоваться в наших собственных целях. Это свойство имеет значение указателя, и мы можем в него вносить любые указатели. Я указываю здесь указатель на структуру *NewQuest*, которая связана с созданным элементом.

Теперь создадим обработчик события *OnChange* для нашего дерева:

```
procedure TQuestionResultForm.QuestionTreeViewChange(Sender: TObject;
  Node: TTreeNode);
var
  i:Integer;
begin
  //Очищаю список
  ResultView.Items.Clear;

  //Если не выделен элемент, то выход
  if Node=nil then exit;

  //Запускаю цикл, по которому заполняются данные списка
  for i:=0 to PQuestion(node.Data).ResultCount-1 do
  with ResultView.Items.Add do
  begin
    Caption:=PQuestion(node.Data).ResiltText[i];
    if PQuestion(node.Data).ResiltValue[i]=true then
    begin
      SubItems.Add('Да');
      ImageIndex:=2;
    end
    else
    begin
      SubItems.Add('Нет');
      ImageIndex:=1;
```

```
end;  
end;  
end;
```

Это событие генерируется каждый раз, когда пользователь выбрал какой-нибудь элемент. По выбору вопроса мы должны заполнить ответы в списке *ListView*. Но прежде чем заполнять, я очищаю список, потому что он уже мог быть заполненным данными другого вопроса.

В обработчик события нам передаётся параметр *Node* типа *TTreeNode*, который указывает на выделенный элемент. Второй строчкой кода я проверяю, если выделенный элемент равен **nil** (ничего не выбрано), то нужно выходить из процедуры.

Чтобы получить доступ к структуре *PQuestion*, в которой хранятся данные об ответах выделенного вопроса, мы должны обратиться к свойству *Data* выделенного элемента *Node*. Как ты помнишь, в это свойство мы поместили указатель на структуру *PQuestion*. Но программа не может знать какого типа этот указатель, поэтому мы должны явно указывать это - *PQuestion(node.Data)*.

Далее, я запускаю цикл от 0 до количества вариантов ответов в данном вопросе *PQuestion(node.Data).ResultCount* минус 1. Внутри цикла я выполняю код *ResultView.Items.Add*, который создаёт очередной элемент списка. Здесь метод *Add* также возвращает указатель на созданный элемент, вместе с которым и будет выполняться дальнейший код (об этом говорит оператор **with**). А внутри кода я выполняю следующее:

1. Заполняю заголовок элемента *Caption*.
2. Если *PQuestion(node.Data).ResultValue[i]* равно *true*, то есть ответ верный, то я добавляю дочерний элемент (текст этого элемента будет отображаться во второй колонке списка) *SubItems.Add('Да')* и присваиваю 2-й индекс иконки. Иначе текст дочернего элемента будет равен «*Нет*» и иконка будет иметь индекс единицы.

Таким образом, внутри цикла будет обработаны все варианты ответов, и все они будут добавлены в список. Попробуй сейчас запустить программу и создать пару вопросов с различными вариантами и посмотреть, как всё будет выглядеть. Моё окно программы ты можешь увидеть на рисунке 25.3.8.

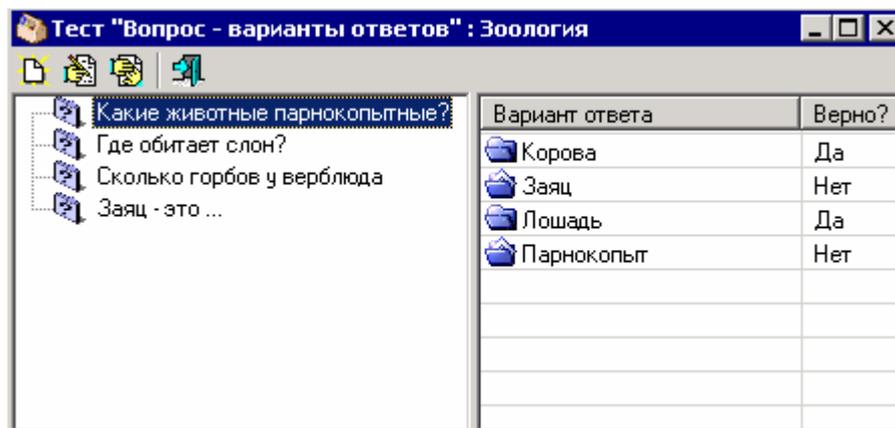


Рисунок 25.3.8 Рабочее окно программы.

Совсем забыл описать код, по которому мы будем отображать окно, показанное мной на рисунке 25.3.8. Для этого нам надо подкорректировать обработчик события *OnClick* для кнопки создания нового проекта теста:

```
procedure TTestEditorForm.NewButtonClick(Sender: TObject);
```

```

begin
NewTestForm.ShowModal;
if NewTestForm.ModalResult<>mrOK then exit;

if NewTestForm.TestTypeBox.ItemIndex=0 then
begin
QuestionResultForm:=TQuestionResultForm.Create(Owner);
QuestionResultForm.ProjectName:=NewTestForm.TestNameEdit.Text;
QuestionResultForm.Caption:=QuestionResultForm.Caption+' : '
+QuestionResultForm.ProjectName;
end;
end;
end;

```

Здесь в первой строке я показываю окно создания нового проекта. Если пользователь выбрал первый тип теста «*Вопрос - варианты ответа*» (в моей книге он будет описан как единственный), то создаётся окно, в котором мы создаём вопросы. Потом я сохраняю имя выбранного проекта и изменяю заголовок окна.

Обработчики события кнопок «*Редактировать*» и «*Удалить*» вопросы я расписывать не буду, а только приведу их код с комментариями. Ты уже должен разобраться с этим кодом:

```

procedure TQuestionResultForm.EditButtonClick(Sender: TObject);
var
i:Integer;
begin
//Здесь QuestionTreeView.Selected указывает на выделенный элемент
//в дереве. Если он равен nil, то ничего не выделено, и нужно выйти
if QuestionTreeView.Selected=nil then exit;

//Заполняю компонент QuestionEdit в окне редактирования вопросов
EditQuestionForm.QuestionEdit.Text:=PQuestion(QuestionTreeView.Selected.Data).Name;

//Очищаю список вариантов ответов в окне редактирования вопросов
EditQuestionForm.ResultListBox.Clear;
for i:=0 to PQuestion(QuestionTreeView.Selected.Data).ResultCount-1 do
begin
//Заполняю список вариантов ответов в окне редактирования вопросов
EditQuestionForm.ResultListBox.Items.Add(
PQuestion(QuestionTreeView.Selected.Data).ResiltText[i]);

//Если ответ верный, то ставлю галочку
if PQuestion(QuestionTreeView.Selected.Data).ResiltValue[i]=true then
EditQuestionForm.ResultListBox.Checked[i]:=true;
end;

//Отображаю окно редактирования вопроса
EditQuestionForm.ShowModal;
if EditQuestionForm.ModalResult<>mrOK then exit;

//Записываю информацию обратно в структуру
PQuestion(QuestionTreeView.Selected.Data).Name:=EditQuestionForm.QuestionEdit.Text;
PQuestion(QuestionTreeView.Selected.Data).ResultCount:=
EditQuestionForm.ResultListBox.Items.Count;
for i:= 0 to PQuestion(QuestionTreeView.Selected.Data).ResultCount-1 do
begin
PQuestion(QuestionTreeView.Selected.Data).ResiltText[i]:=
EditQuestionForm.ResultListBox.Items.Strings[i];
PQuestion(QuestionTreeView.Selected.Data).ResiltValue[i]:=

```

```

EditQuestionForm.ResultListBox.Checked[i];
end;

//Вызываю процедуру QuestionTreeViewChange, которая должна обновить
//информацию в ResultView. Первый параметр нас не интересует, а второй
//мы обязаны указать, потому что внутри процедуры QuestionTreeViewChange
//мы используем его. Я указываю выделенный элемент.
QuestionTreeViewChange(nil, QuestionTreeView.Selected);
end;

```

Единственное, что я здесь хочу отметить, так это то, что здесь я обращаюсь к структуре связанной с элементом через свойство *Data* выделенного элемента. Как я уже говорил, там храниться указатель на структуру. То же самое можно было бы делать, обращаясь через контейнер, просто в данном случае это будет не так удобно. Но всё же это возможно и на всякий случай я приведу пример, как можно обратиться к свойству *Name*:

```

PQuestion(QuestionList[QuestionTreeView.Selected.Index]).Name

```

Здесь я использую контейнер *QuestionList*. В квадратных скобках у него я указываю индекс элемента из контейнера, который мне нужен. Здесь я указываю индекс выделенного в дереве элемента *QuestionTreeView.Selected.Index*.

По нажатию кнопки «Удалить» ты должен написать следующий код:

```

procedure TQuestionResultForm.DeleteButtonClick(Sender: TObject);
var
  index, i:Integer;
begin
  if QuestionTreeView.Selected=nil then exit;

  //Подтверждение удаления
  if Application.MessageBox(PChar('Вы действительно хотите удалить - '+
    QuestionTreeView.Selected.Text), 'Внимание!!!',
    MB_OKCANCEL+MB_ICONINFORMATION)<>idOk then Exit;

  //Сохраняю индекс выделенного элемента
  index:=QuestionTreeView.Selected.Index;

  //Удаляю выделенный элемент из дерева
  QuestionTreeView.Items.Delete(QuestionTreeView.Selected);

  //Удаляю из контейнера
  QuestionList.Delete(Index);
end;

```

 На компакт диске, в директории \Примеры\Глава 25\Test2\Редактор ты можешь увидеть исходник уже написанного примера.

25.4. Сохранение и загрузка теста.

Твоя программа уже умеет создавать тесты, пора бы её научить и сохранять их и тем более загружать потом созданные проекты для редактирования. Я вынес кнопки открытия и сохранения проекта из дочернего окна в основное. Если честно, то сохранение легче сделать внутри дочернего окна, а открытие в главном. Но я не пошёл простым путём, потому что хочу тебе показать, как работать с дочерними окнами.

Итак, по нажатию кнопки «Сохранить» проект пишем следующий код:

```
procedure TTestEditorForm.SaveButtonClick(Sender: TObject);
begin
  //Если активное дочернее окно равно нулю
  //(нет активных окон), то выход
  if ActiveMDIChild=nil then exit;
  //Если окно имеет имя QuestionResultForm, то это
  //вопрос-варианты ответов и вызываем для сохранения
  //процедуру SaveTest1.
  if ActiveMDIChild.Name='QuestionResultForm' then
    SaveTest1;
end;
```

Свойство *ActiveMDIChild* всегда указывает на активное в данный момент дочернее окно. Прежде чем использовать это свойство, его желательно сравнивать со значением **nil**, потому что в данный момент может вообще не быть ни одного дочернего окна. В этом случае, при обращении к свойству может произойти критическая ошибка.

Процедура *SaveTest1* должна выглядеть следующим образом:

```
procedure TTestEditorForm.SaveTest1;
var
  fs:TFileStream;
  i:Integer;
  Str:String[5];
begin
  //Если у активного окна в свойстве FileName пусто,
  //то нет имени файла и нужно вызвать обработчик события
  //меню "Сохранить как...", чтобы появилось окно ввода
  //имени файла
  if TQuestionResultForm(ActiveMDIChild).FileName="" then
    begin
      SaveAsMenuClick(nil);
      exit;
    end;

  //Создаю новый файл. Если он уже существовал, то его
  //содержимое будет уничтожено
  fs:=TFileStream.Create(TQuestionResultForm(ActiveMDIChild).FileName, fmCreate);

  //Сохраняю в начале файла текст "Тест", чтобы по нему потом
  //определить к чему относиться данный файл.
  Str:='Тест';
  fs.Write(Str, SizeOf(Str));

  //Сохранить имя проекта
  fs.Write(TQuestionResultForm(ActiveMDIChild).ProjectName,
    sizeof(TQuestionResultForm(ActiveMDIChild).ProjectName));
  try
    //Сохранить количество вопросов
    fs.Write(TQuestionResultForm(ActiveMDIChild).QuestionList.Count,
```

```

        sizeof(TQuestionResultForm(ActiveMDIChild).QuestionList.Count));

    //Запускаю цикл, в котором сохраняются все вопросы.
    for i:=0 to TQuestionResultForm(ActiveMDIChild).QuestionList.Count-1 do
        fs.Write(PQuestion(TQuestionResultForm(ActiveMDIChild).QuestionList[i])^,
            sizeof(TQuestion));
    finally
        //Закрывать файл
        fs.Free;
    end;
end;

```

Здесь всё очень просто и с кодом можно разобраться по комментариям. Единственное, на что я хочу обратить внимание – наша структура *PQuestion* находится в динамической памяти, поэтому при сохранении нужно указывать знак разыменования \wedge . Если этого знака не указать, то в файл сохраниться адрес структуры, а не сама структура. В этом случае при чтении данных из файла мы прочитаем адрес, но по этому адресу ничего хорошего не будет, потому что после первой же перезагрузки программы память очиститься и сама структура уничтожится. Поэтому, для сохранения данных по адресу, а не самого адресу нужно указывать знак \wedge .

Обработчик события для пункта меню «*Сохранить как...*» ещё проще:

```

procedure TTestEditorForm.SaveAsMenuClick(Sender: TObject);
begin
    if SaveDialog1.Execute then
        begin
            TQuestionResultForm(ActiveMDIChild).FileName:=SaveDialog1.FileName;
            SaveButtonClick(nil);
        end;
end;

```

Здесь я отображаю окно выбора имени файла. Если пользователь что-то выбрал, то сохраняю имя файла в свойстве *FileName* активного окна, и вызываю обработчик события кнопки «*Сохранить*», где происходит сохранение.

Теперь посмотри на обработчик события *OnClick* для кнопки «Открыть» проект:

```

procedure TTestEditorForm.OpenButtonClick(Sender: TObject);
var
    fs:TFileStream;
    i, Count:Integer;
    Str:String[5];
    NewQuest:PQuestion;
begin
    //Показать окно открытия файла
    if not OpenFileDialog1.Execute then exit;

    //Открыть файл для чтения
    fs:=TFileStream.Create(OpenDialog1.FileName, fmOpenRead);

    //Перейти в начало файла и прочитать заголовок
    fs.Seek(0,soFromBeginning);
    fs.read(Str, SizeOf(Str));

    //Если заголовок равен тексту "Тест", значит это "вопрос-

```

```

//варианты ответов".
if Str='Тест' then
begin
//Создать новое окно теста
QuestionResultForm:=TQuestionResultForm.Create(Owner);

//Сохранить имя открытого файла в объекте окна
QuestionResultForm.FileName:=OpenDialog1.FileName;

//Прочитать имя проекта
fs.Read(QuestionResultForm.ProjectName, sizeof(QuestionResultForm.ProjectName));

try
//Прочитать количество вопросов
fs.Read(Count, sizeof(Count));

//Запустить цикл чтения вопросов
for i:=0 to Count-1 do
begin
//Создаю новую структуру в памяти для вопроса
NewQuest:=New(PQuestion);
//Читаю структуру
fs.Read(NewQuest^, sizeof(TQuestion));

//Добавляю структуру в контейнер
QuestionResultForm.QuestionList.Add(NewQuest);

//Создаю новый элемент в дереве
with QuestionResultForm.QuestionTreeView.Items.Add(nil, NewQuest.Name) do
begin
ImageIndex:=0;
Data:=NewQuest;
end;
end;
finally
//Закрываю файл
fs.Free;
end;
end;
end;

```

В чтении файла так же ничего сложного нет. Всё очень похоже на запись и со всеми методами ты уже должен быть знаком. Здесь так же мы читаем данные в указатель на структуру *PQuestion*, поэтому при чтении нужно разыменовывать указатель *NewQuest^*, чтобы данные записались «по адресу», а не в адрес.

 На компакт диске, в директории \Примеры\Глава 25\Test3\Редактор ты можешь увидеть исходник уже написанного примера.

Вот на этом наш редактор можно считать законченным. Хотя ещё не реализованы обработчики события для кнопок печати и свойств проекта. Но свойства проекта нам не нужны, а вот печать я оставлю тебе. Попробуй сам добавить вывод на печать нашего проекта.

25.5. Тестер.

Теперь напишем программу тестирования, которая будет загружать наши проекты, отображать вопросы и собирать статистику правильных ответов. Для этого у нас будет отдельная программа, поэтому создай новый проект и установи на форму следующие компоненты (мою форму ты можешь увидеть на рисунке 25.5.1):

1. Панель *ToolBar* с тремя кнопками «Открыть», «Запустить» и «Выход».
2. Компонент *StaticText*, где будем отображать вопросы. В свойстве *Name* укажи *QuestionLabel* и свойство *AutoSize* установи в *false*.
3. Список *CheckListBox* в котором будут отображаться варианты ответов. В свойстве *Name* укажи *QuestionCheckList*.
4. Ну и последнее - кнопку «Дальше».

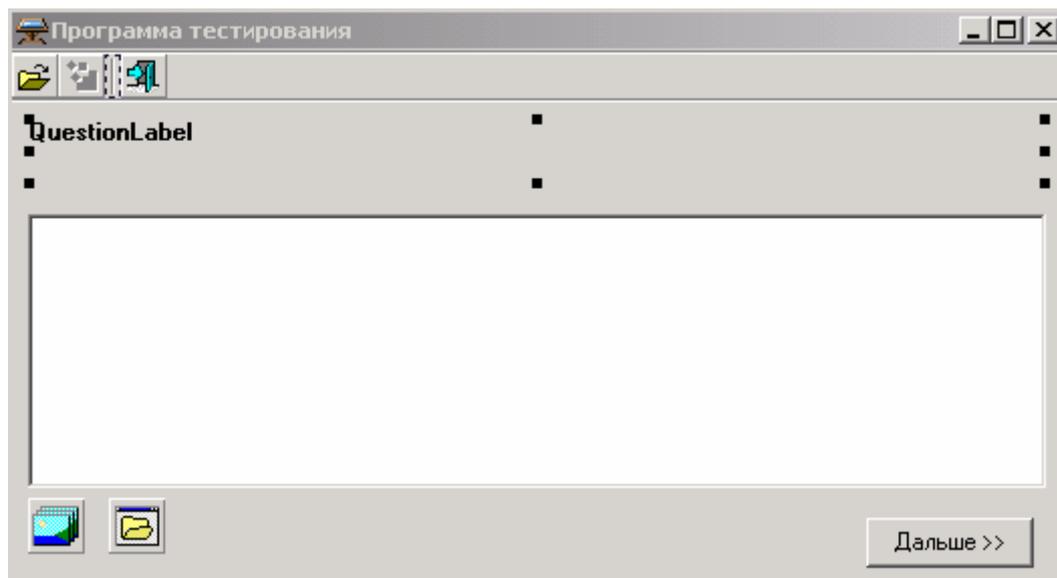


Рисунок 25.5.1 Форма будущей программы.

В разделе **type** объяви структуру *TQuestion*, такого же вида, как и в редакторе вопросов. Количество и размерность полей структуры должно быть одинаково, потому что мы будем использовать её для загрузки данных из файла. Если хоть какое-то поле будет отличаться, то при загрузке данных произойдёт ошибка.

```
type
PQuestion=^TQuestion;
TQuestion=record
Name: String[255];
ResultCount:Integer;
ResiltText: array[0..10] of String[255];
ResiltValue: array[0..10] of boolean;
end;
```

В разделе **private** объяви следующие переменные:

```
private
{ Private declarations }
QuestionList:TList;
Question, QuestionNumber, FalseNumber:Integer;
```

FileName:String;

Разберём, для чего нужны эти переменные:

QuestionList – здесь будет храниться список вопросов, как и у редактора вопросов.

Question – будет отображать текущий вопрос, на который отвечает испытуемый.

QuestionNumber – здесь мы будем хранить количество вопросов, на которые уже даны ответы. Нам же надо иметь счётчик, после которого тест должен закончиться.

FalseNumber – количество неправильных ответов.

Теперь создадим обработчик события *OnShow* для главной формы. В этом обработчике нужно инициализировать список *QuestionList*:

```
procedure TTestForm.FormShow(Sender: TObject);
begin
  QuestionList:=TList.Create;
end;
```

По событию *OnDestroy* мы должны уничтожить этот объект:

```
procedure TTestForm.FormDestroy(Sender: TObject);
begin
  QuestionList.Free;
end;
```

Теперь для кнопки открытия пишем следующий код:

```
procedure TTestForm.OpenButtonClick(Sender: TObject);
begin
  //Показать окно открытия файла
  if not OpenDialog1.Execute then exit;
  FileName:=OpenDialog1.FileName;
  RunButton.Enabled:=true;
end;
```

В первой строке я отображаю окно открытия файла. Если пользователь нажал на кнопку «Отмена», то происходит выход из процедуры. Иначе, в переменной *FileName* сохраняется имя выбранного файла. В принципе, этого можно было и не делать, потому что имя файла хоть как останется в свойстве *OpenDialog1.FileName*, но я всё же завёл отдельную переменную, и буду хранить имя файла там.

В последней строке я делаю кнопку «Запустить» *RunButton* доступной. Кстати, на форме эта кнопка должна быть не доступной, чтобы при старте программы, пользователь не мог нажать кнопку «Запустить», пока не выберет файл.

Теперь пишем обработчик события *OnClick* для кнопки «Запустить»:

```
procedure TTestForm.RunButtonClick(Sender: TObject);
begin
```

```
LoadFile;  
QuestionNumber:=0;  
FalseNumber:=0;  
NextButton.Enabled:=true;  
NextQuestion;  
end;
```

В первой строке я вызываю процедуру *LoadFile*, которую я напишу чуть позже, и она будет загружать список вопросов из выбранного файла проекта. Почему я должен загружать вопросы каждый раз при старте программы? Да потому что тест будет происходить следующим образом:

1. Из списка вопросов случайным образом выбирается первый попавшийся вопрос.
2. Пользователь отвечает на него, и мы удаляем его из списка. Таким образом, в следующий раз, когда мы будем выбирать вопрос из списка, то мы уже точно уверены, что в списке нет вопроса, на который бы уже отвечал пользователь.
3. При следующем старте теста список вопросов инициализируется заново (мы снова загружаем весь список) и все вопросы возвращаются на свои места.

После загрузки вопросов я обнуляю все переменные, и делаю доступной кнопку *NextButton* (это кнопка «Дальше», по нажатию которой будет выбираться следующий вопрос). При старте программы кнопка «Дальше» должна быть недоступной.

В последней строке я вызываю процедуру *NextQuestion*, которая и будет выбирать случайный вопрос и отображать его в окне программы.

Теперь посмотрим на процедуру загрузки вопросов *LoadFile*. Она идентична уже написанной процедуре загрузки в программе редактора вопросов:

```
procedure TTestForm.LoadFile;  
var  
  fs:TFileStream;  
  i, Count:Integer;  
  Str:String[5];  
  ProjectName:String[255];  
  NewQuest:PQuestion;  
begin  
  QuestionList.Clear;  
  //Открыть файл для чтения  
  fs:=TFileStream.Create(FileName, fmOpenRead);  
  
  //Перейти в начало файла и прочитать заголовок  
  fs.Seek(0,soFromBeginning);  
  fs.read(Str, SizeOf(Str));  
  
  //Если заголовок равен тексту "Тест", значит это "вопрос-  
  //варианты ответов".  
  if Str='Тест' then  
    begin  
      //Прочитать имя проекта  
      fs.Read(ProjectName, sizeof(ProjectName));  
      Caption:=ProjectName;  
  
      try  
        //Прочитать количество вопросов  
        fs.Read(Count, sizeof(Count));  
  
        //Запустить цикл чтения вопросов  
        for i:=0 to Count-1 do  
          begin
```

```

//Создаю новую структуру в памяти для вопроса
NewQuest:=New(PQuestion);
//Читаю структуру
fs.Read(NewQuest^, sizeof(TQuestion));

//Добавляю структуру в контейнер
QuestionList.Add(NewQuest);
end;
finally
//Закрываю файл
fs.Free;
end;
end;
end;

```

Тут всё должно быть понятно, но я на всякий случай снабдил весь код подробными комментариями.

Теперь посмотрим на процедуру *NextQuestion*, которая должна случайным образом выбирать вопрос из списка:

```

procedure TTestForm.NextQuestion;
var
i:Integer;
begin
Randomize;
Question:=Random(QuestionList.Count-1);

QuestionLabel.Caption:=PQuestion(QuestionList[Question]).Name;

QuestionCheckList.Items.Clear;
for i:=0 to PQuestion(QuestionList[Question]).ResultCount-1 do
QuestionCheckList.Items.Add(PQuestion(QuestionList[Question]).ResiltText[i]);

Inc(QuestionNumber);
end;

```

В первой строке процедуры я вызываю процедуру *Randomize*, которая инициализирует таблицу случайных чисел. Если ты опустишь вызов этой процедуры, то ничего страшного не произойдёт, и когда ты будешь запрашивать случайное число, то оно будет случайным, но всё же лучше инициализировать таблицу. Просто в этом случае считается, что случайность будет более случайной (во как звучит!!!).

Во второй строке я вызываю функцию *Random*, которая возвращает случайное число. Ей нужно передать в качестве параметра максимально допустимое число. Я передаю *QuestionList.Count-1*, т.е. количество вопросов в нашем списке. Функция вернёт мне случайное число от 0 до указанного числа. Я сохраняю это число в переменной *Question*.

В следующей строке кода я показываю в компоненте *QuestionLabel* вопрос соответствующий вопрос. Затем очищаю список ответов в компоненте *QuestionCheckList* и заполняю его вариантами ответов, относящихся к данному вопросу. В последней строке кода я увеличиваю переменную *QuestionNumber*, в которой у нас храниться количество отвеченных вопросов.

По нажатию кнопки «Далее» пишем следующий код:

```

procedure TTestForm.NextButtonClick(Sender: TObject);
var
  OK:Boolean;
  i:Integer;
begin
  OK:=true;

  for i:=0 to PQuestion(QuestionList[Question]).ResultCount-1 do
    if PQuestion(QuestionList[Question]).ResiltValue[i]<>QuestionCheckList.Checked[i] then
      OK:=false;

  if OK=false then
    Inc(FalseNumber);

  //Удаление вопроса из списка
  QuestionList.Delete(Question);

  if QuestionNumber<5 then
    NextQuestion
  else
    begin
      Application.MessageBox(PChar('Вы закончили тест с количеством ошибок = '+
        IntToStr(FalseNumber)), 'Внимание!!!');
      NextButton.Enabled:=false;
    end;
end;

```

В первой строке здесь устанавливается логическая переменная *OK* в значение *true*. В этой переменной мы будем хранить состояние результата ответа. По умолчанию будем считать, что ответ правильный, поэтому и устанавливаем значение *true*.

Далее, запускаю цикл от 0 до количества вариантов ответов в списке. Внутри цикла я сравниваю значение правильных ответов с состоянием свойство *Checked* компонента *QuestionCheckList*. Если хоть что-то не совпадает, то тестируемый где-то ошибся и нужно установить переменную *OK* в значение *false*, т.е. ответ неверный. После цикла происходит проверка, если переменная *OK* равна *false*, то увеличиваем счётчик неправильных ответов *FalseNumber* на единицу.

Всё, текущий вопрос нам больше в списке не нужен, и его нужно удалить, чтобы он больше не появился, когда мы будем случайным образом получать следующий вопрос.

Дальше происходит проверка, если количество отображённых вопросов меньше 5, то выбираем следующий вопрос (вызываем процедуру *NextQuestion*), иначе отображаем сообщение с состоянием пройденного теста и делаем кнопку «Далее» недоступной.

Как видишь, мой тест состоит из 5 вопросов, если тебе нужно больше, то можешь увеличить это значение. Но у нас в редакторе вопросов есть кнопка свойств, по нажатию которой можно отображать окно свойств проекта. Я бы сделал возможность в этом окне выбирать количество вопросов, на которые должен ответить испытуемый. Потом эти свойства можно сохранить в файл проекта и загружать в нашей программе теста. Но всё это я делать не буду, потому что у тебя уже есть достаточно знаний, чтобы попробовать всё это сделать самому.

 На компакт диске, в директории \Примеры\Глава 25\Test4\ ты можешь увидеть исходник уже написанного примера.