№ 28 (221) июль 1999

еженедельное
ПРИЛОЖЕНИЕ К ГАЗЕТЕ
«ПЕРВОЕ СЕНТЯБРЯ»

ИНФОРМАТИК

Программирование на языке Visual Basic 5

А.И. Маргулев

Выпуск 2

Жаркое 99 лето

Содержание

Выбор и размещение элементов управления

Продолжение

Займемся теперь написанием кода обработчиков событий (кроме очевидного для кнопки выхода), связанных с элементами управления нашей программы и отвечающих разработанному сценарию.

❐ Каждое нечетное нажатие на кнопку **cmdBrows** приводит к активизации броузера, четное — к его "закрытию". Соответствующие ветви инструкции **If** выбираются путем сравнения с 0 остатка от деления на 2 значения счетчика (переменная **i**) числа нажатий на эту кнопку.

```
Private Sub cmdBrows_Click()
 Static i
 i = i + 1
 If i Mod 2 = 1 Then
'Делаем видимыми элементы броузера,
  Drive1.Visible = True
  Dir1.Visible = True
  File1.Visible = True
  Image1.Visible = False 'а отображение графики отключаем
  cmdBrows.Caption = "OK"
 Else
  Drive1.Visible = False 'После настройки - наоборот
  Dir1.Visible = False
  File1.Visible = False
  Image1.Visible = True
  'Генерируем случайный индекс:
  li = li_o = Int(lsbRus.ListCount * Rnd)
  'Отображаем соответствующее ему русское слово:
  lblWord.Caption = lsbRus.List(li)
  'И картинку:
  Image1.Picture = LoadPicture(File1.Path + "\" + _
         Mid(Str(lsbRus.ItemData(li)), 2) + ".jpg")
  cmdBrows.Caption = "Обзор..."
 End If
End Sub
```

❐ Уже в первом фрагменте мы использовали две глобальные переменные: **li** и **li_o**. Первая хранит индекс текущего, случайным образом выбранного в массиве **lsbRus.List** русского слова, вторая — предшествующего. Хранить предшествующий индекс нужно для предотвращения ситуации, когда второй раз подряд выпадает одно и то же слово; в таком случае мы будем брать соседнее слово (это самый примитивный из способов избежать повторов).

Приведем всю секцию **(General)**, содержащую описание всех глобальных переменных программы:

```
Dim li As Integer, li_o As Integer, t As Integer, _
    n As Integer, nf As Integer
' li   - индекс очередного элемента в списке русских слов
' li_o - индекс предыдущего -//-
' t    - счетчик секунд теста
' n    - число ответов во время теста
' nf   - из них правильных
```

- В следующем фрагменте приведены коды обработчиков событий, отражающих работу элементов-списков броузера:

```
Private Sub Drive1_Change()
 Dir1.Path = Drive1.Drive 'Свойство Drive - текущий диск,
End Sub
'Path - текущий каталог на текущем диске

Private Sub Dir1_Change()
 File1.Path = Dir1.Path
 lblPath.Caption = Dir1.Path 'Отображаем текущий путь
End Sub

Private Sub File1_Click()
   Image1.Visible = True
 Image1.Picture = LoadPicture(File1.Path + "\" + _
         File1.filename)
End Sub
```

- Обработка выбора (событие **Click**) из списка английских слов-переводов в ответ на появление очередного случайно выбранного русского слова (после предшествующего выбора либо после первоначальной установки рабочего каталога):

```
Private Sub cboEngl_Click()
 n = n + 1
 If cboEngl.ItemData(cboEngl.ListIndex) = _
                                      lsbRus.ItemData(li)
Then
   MMControl1.filename = File1.Path + "\" + _
              Mid(Str(lsbRus.ItemData(li)), 2) + ".wav"
 Else
   lblFault.Visible = True
    'Ошибку сопровождает звуковой сигнал:
  MMControl1.filename = File1.Path + "\" + "ugly.wav"
  nf = nf + 1
 End If
 'Запускаем таймер tmrDelay:
 tmrDelay.Enabled = True
 'Команда "sound" определена для устройства WaveAudio и
 'не требует его предварительного открытия:
 MMControl1.Command = "sound"
End Sub
```

- Запущенный обработчиком выбора из списка **cboEngl** таймер **tmrDelay** через заданный промежуток времени (2 секунды) обеспечит очередной случайный выбор русского слова из массива **lsbRus.List**:

```
Private Sub tmrDelay_Timer()
 li = Int(lsbRus.ListCount * Rnd)
 'Если очередной индекс совпал с предыдущим:
 If li = li_o Then
  If li = lsbRus.ListCount - 1 Then
   li = li - 1
  Else
   li = li + 1
  End If
 End If
 'Отображаем случайно выбранное слово:
 lblWord.Caption = lsbRus.List(li)
 'И соответствующую картинку:
 Image1.Picture = LoadPicture(File1.Path + "\" + _
        Mid(Str(lsbRus.ItemData(li)), 2) + ".jpg")
 'А сам таймер после этого останавливаем:
 tmrDelay.Enabled = False
 'И считаем, что очередная итерация стала уже предыдущей:
 lblFault.Visible = False
 li_o = li
End Sub
```

- Нажатие на кнопку запуска теста **cmdStart** запускает таймер **tmrSec**, в обработчике которого переменная-счетчик отсчитывает время теста, выдаваемое в надписи на кнопке **cmdStart**:

```
Private Sub tmrSec_Timer()
 t = t + 1
 cmdStart.Caption = Str(t) + " сек."
End Sub
```

- Обработчик нажатия на кнопку запуска теста **cmdStart**, кроме запуска таймеров **tmrSec** и **tmrTest**, делает первую (в рамках теста) генерацию случайно выбранного русского слова и соответствующим образом его отображает:

```
Private Sub cmdStart_Click()
 t = 0 'Обнуляем счетчик секундомера
 n = 0 'Количество ответов во время теста
 nf = 0 'Из них правильных
 Drive1.Visible = False
 Dir1.Visible = False
 File1.Visible = False
```

```
li = Int(lsbRus.ListCount * Rnd)
 If li = li_o Then
  If li = lsbRus.ListCount - 1 Then
   li = li - 1
  Else
   li = li + 1
  End If
 End If
 lblWord.Caption = lsbRus.List(li)
 Image1.Visible = True
 Image1.Picture = LoadPicture(File1.Path + "\" + _
 Mid(Str(lsbRus.ItemData(li)), 2) + ".jpg")
 tmrSec.Enabled = True
 tmrTest.Enabled = True
 li_o = li
End Sub
```

- Таймер **tmrTest** генерирует через 60 секунд после своего запуска кнопкой **cmdTest** событие **Timer**. В его обработчике с помощью инструкции выбора **Select Case** организуется расчет оценки тестируемого исходя из количества данных им ответов и количества правильных из них.

  Инструкция **Select Case** имеет следующую структуру:

```
Select Case Выражение
  Case Диапазон_1
    Блок_1
  Case Диапазон_2
    Блок_2
.......................
  Case Диапазон_n
    Блок_n
  Case Else
    Блок_default
End Select
```

  В приведенной структуре можно выделить граничные "скобки" инструкции, состоящие из строк заголовка **Select Case** и окончания инструкции **End Select**, и **Case**-ветви, каждая из которых состоит из строки заголовка, начинающегося словом **Case**, и следующего за ним блока инструкций. При этом ни одна **Case**-ветвь или ее блок не являются обязательными. После слова **Case** заголовка **Case**-ветви всегда имеется некоторый список (возможно, состоящий из одного элемента) выражений или специальных конструкций, который задает множество значений (диапазон) числовой прямой (отдельные точки, сплошные отрезки и лучи).

Мы будем рассматривать данную инструкцию, предполагая, что входящие в заголовки выражения имеют числовой тип. Это, однако, не является обязательным, поскольку операции сравнения определены и для строк (а некоторые — и для объектов).

Исполнение инструкции происходит по следующему алгоритму:

- Вычисляется ***Выражение*** в заголовке инструкции.
- Последовательно сверху вниз проверяются заголовки **Case**-ветвей, содержащие ***Диапазоны***, на вхождение вычисленного значения ***Выражения*** в эти ***Диапазоны***. Как только *первая* такая **Case**-ветвь найдена, выполняется ее ***Блок*** (хотя бы и пустой).
- Если значение ***Выражения*** не входит ни в один ***Диапазон*** и в инструкции имеется ветвь **Case Else**, то выполняется ее ***Блок***.
- После выполнения ***Блока*** какой-либо **Case**-ветви либо при невыполнении ни одного ***Блока*** инструкция завершается.

Для тех, кому пока не хочется отвлекаться на изучение приложения, перечислим, чем может являться каждый элемент списка, задающего какой-либо ***Диапазон***.

Им может быть, во-первых, любое выражение (константа, переменная, обращение к функции, любые комбинации перечисленных объектов, соединенных допустимым образом знаками операций и взятием подобных соединений в качестве аргументов обращений к функциям).

Во-вторых, таким элементом может быть конструкция

Выражение_1 To Выражение_2,

где ***Выражение_1*** и ***Выражение_2*** — соответственно начальное и конечное значения отрезка, задающего множество значений, входящих в итоговый ***Диапазон***.

В третьих, конструкция

Is *Оператор Выражение*

где ***Оператор*** — один из 6 операторов сравнения:

<	(меньше)
<=	(меньше либо равно)
>	(больше)
>=	(больше либо равно)
=	(равно)
<>	(не равно)

Выражение вычисляется, и его значение вместе с оператором задают множество значений, входящих в итоговый ***Диапазон*** (например, < 5 задает множество всех значений, меньших, чем 5).

```
Private Sub tmrTest_Timer()
 Dim e As Integer
 cmdStart.Caption = "Запуск теста (60 сек.)"
 tmrDelay.Enabled = False
 tmrSec.Enabled = False
 lblFault.Visible = False
 Select Case n - nf
  Case Is > 9
   If nf = 0 Then
    e = 5
   Else
    e = 4
   End If
  Case 7 To 9
   If nf = 0 Then
    e = 4
   Else
    e = 3
 End If
  Case 4 To 6
   If nf = 0 Then
    e = 3
   Else
    e = 2
   End If
  Case Else
   e = 2
 End Select
 MsgBox "Дано " & n & " ответов; число Ваших ошибок:" & _
    nf & Chr(13) & "Отметка: " & e, vbInformation,"ТЕСТ"
 tmrTest.Enabled = False
End Sub
```

В приведенном обработчике можно отметить также обращение к функции **MsgBox**, которая выводит на экран заданную программистом информацию в специальном окне сообщений **(Message Box)**. Окно сообщений является диалогом, причем модальным: программа (либо даже операционная система) не будет продолжать работу до тех пор, пока пользователь не щелкнет по какой-либо кнопке, расположенной в окне диалога.

Обращение к данной функции имеет, во-первых, стандартную синтаксическую форму, подразумевающую обязательное использование такого обращения в составе выражения:

MsgBox (*Текст_сооб., Признак, Текст_заг., Имя_файла_спр., Номер_разд.*),

а во-вторых, форму обращения к методу (без скобок, в отдельной строке программы):

MsgBox ***Текст_сооб., Признак, Текст_заг., Имя_файла_спр., Номер_разд.***

— в тех случаях, когда возвращаемое функцией значение заведомо не используется. На *рис.* 3.2 приведено строение окна сообщения, выдаваемое вызовом функции **MsgBox** в нашем обработчике.

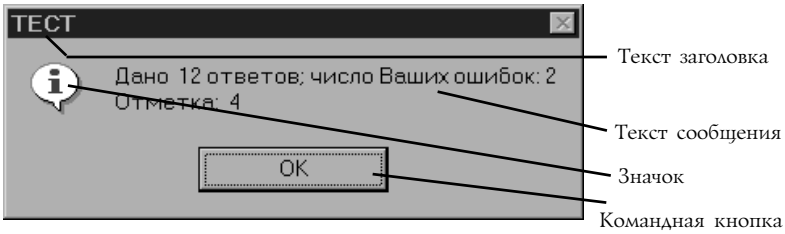


Рис. 3.2. Элементы окна сообщения

В приведенных формах обращения обязательным является наличие только первого аргумента ***(Текст_сообщения)***, представляющего строковое выражение, содержащее выводимое сообщение. Если других аргументов нет, то по умолчанию имя окна сообщения будет **Project1**, значок отсутствует и имеется единственная кнопка **OK** (при нажатии на которую функция возвращает значение 1, если обращение входит в состав выражения). Если в списке аргументов между парой присутствующих аргументов имеются опущенные, то следующие после каждого из опущенных аргументов запятые не опускаются (это общее правило для любого обращения к функции с переменным числом аргументов). Третий аргумент ***(Текст_заголовка)*** задает строковым выражением наименование окна сообщения; 4-й и 5-й — файл справки (о данном окне) и номер раздела в нем. Наконец, второй аргумент — ***Признак*** — определяет 4 независимые характеристики окна сообщений: наличие и тип значка, набор кнопок, какая кнопка имеет начальный фокус, тип модальности диалога. Рассмотрим этот аргумент подробнее.

Для каждой из указанных характеристик существует таблица альтернативных значений. Выбранные по одному из каж-

дой таблицы значения-слагаемые складываются, давая уникальное значение аргумента ***Признак***. В табл. 3.1 приведены слагаемые, определяющие наличие и тип используемого в окне сообщения значка.

Таблица 3.1. Значки и определяющие их значения

Значок	Наименование значка	Слагаемое в аргумент *Признак*	Назначение
	Критическая ситуация	16	Указание на серьезную, обычно фатальную ошибку
	Запрос	32	Требование дополнительной информации для продолжения работы
	Предупреждение	48	Указание на возможность ошибки
	Информационное сообщение	64	Сообщение пользователю о состоянии выполнения программы

А сочетание имеющихся в окне кнопок задает следующая таблица:

Слагаемое в аргумент *Признак*	Отображаемый набор кнопок
0	Отображается только кнопка “OK"
1	Отображаются кнопки “OK” и “Отмена” (Cancel)
2	Отображаются кнопки “Прервать”(Abort),“Повторить” (Retry) и “Пропустить” (Ignore)
3	Отображаются кнопки “Да” (Yes), “Нет” (No) и “Отмена” (Cancel)
4	Отображаются кнопки “Да” (Yes) и “Нет” (No)
5	Отображаются кнопки “Повторить” (Retry) и “Отмена” (Cancel)

Определение первоначального фокуса задает таблица:

Слагаемое в аргумент *Признак*	**Кнопка, на которую первоначально направлен фокус**
0	Первая
256	Вторая
512	Третья
768	Четвертая

Наконец, таблица типа модальности содержит всего два значения: 0 — если окно модально для приложения и 4096 — если его модальность распространяется на всю систему.

Заметим, что каждое из приведенных в таблицах значений слагаемых можно использовать в форме именованной константы (все они приведены в аналогичных таблицах справки по функции **MsgBox**), имя которой мнемонически указывает назначение данного слагаемого. Использование именованных констант в данном случае если и не облегчает программирование (их слишком много для запоминания), то облегчает последующее чтение кода.

Наконец, приведем таблицу возвращаемых функцией значений, определяемых выбором нажатия той или иной кнопки:

Кнопка	OK	Cancel	Abort	Retry	Ignore	Yes	No
Значение	1	2	3	4	5	6	7

- Загрузка формы **ТЕСТ** сопровождается и начальной генерацией отображений случайно выбранного русского слова — в том случае, если рабочим каталогом сразу оказывается каталог _PIC-WAV. Проверку вхождения строки "_PIC-WAV" в текущий путь, заданный свойством **Dir1.Path**, осуществляет функция **Instr**(***Строка_1, Строка_2***), где первый аргумент — строка, в которой ведется проверка, второй — строка, проверяемая на вхождение (могут задаваться еще два необязательных аргумента, информацию о которых можно получить из справки):

```
Private Sub Form_Load()
 Dim s As String
 Randomize
 Drive1.Visible = False
 Dir1.Visible = False
 File1.Visible = False
 lblPath.Caption = Dir1.Path
 s = Dir1.Path
 If InStr(s, "_PIC_WAV") <> 0 Then
  Image1.Visible = True
  li = li_o = Int(lsbRus.ListCount * Rnd)
  lblWord.Caption = lsbRus.List(li)
  Image1.Picture = LoadPicture(File1.Path + "\" + _
           Mid(Str(lsbRus.ItemData(li)), 2) + ".jpg")
 End If
End Sub
```

В остальном все элементы этого обработчика аналогичны уже рассмотренным.

Возможно, у вас давно уже возник вопрос: откуда можно брать файлы с картинками и как записать "сопровождающие" их звуковые файлы? Наиболее естественно начинать создание нашей базы данных именно с подбора картинок, отображающих предметы, соответствующие уровню сложности формируемого словаря. Огромные наборы подходящих картинок можно найти на компакт-дисках (CD). Лучше всего, если это будут растровые картинки (форматы BMP и PCX), картинки в виде метафайлов (WMF и EMF) либо картинки в форматах JPEG и GIF (иначе их придется переводить в один из этих форматов программой типа Corel PhotoPaint). (Забегая вперед, скажем, что в "настоящей" БД картинки должны быть только в форматах BMP и PCX, что, безусловно, является недоработкой разработчиков языка.) Звуковые файлы записываются (посредством микрофона, подключенного к звуковой карте) с помощью приложения **Фонограф** (Sound Recorder), находящегося в группе **Мультимедиа**, входящей в группу **Стандартные** из пункта **Программы** пускового меню Windows 95.

Вот как выглядит окно созданного приложения в процессе тестирования обучаемого:

Первый вариант "Словарной обучалки" создан!

Ну а если у нас "настоящая" база данных?

Давайте представим, что мы тем или иным способом сумели создать и заполнить информацией БД. Как нам с ней дальше работать?

Мы уже упоминали о семействе объектов DAO, предназначенных именно для этой цели. Но, оказывается, в VB5 есть еще ряд средств, сильно облегчающих решение наших проблем.

Для организации работы с уже имеющейся базой данных VB5 содержит, во-первых, специальный компонент — элемент управления данными (**Data Control**, в подсказке — просто **Data**), а во-вторых — ряд элементов управления (всего их 15), выполняющих по отношению к нему функцию "связанных элементов управления" *(bound control)*. Связанные элементы управления — это в основном стандартные компоненты, имеющие ряд дополнительных свойств, обеспечивающих доступ к базе данных под управлением элемента **Data**. На *рис.* 3.3 приведена палитра инструментов с указанием как элемента **Data**, так и большинства связанных элементов.

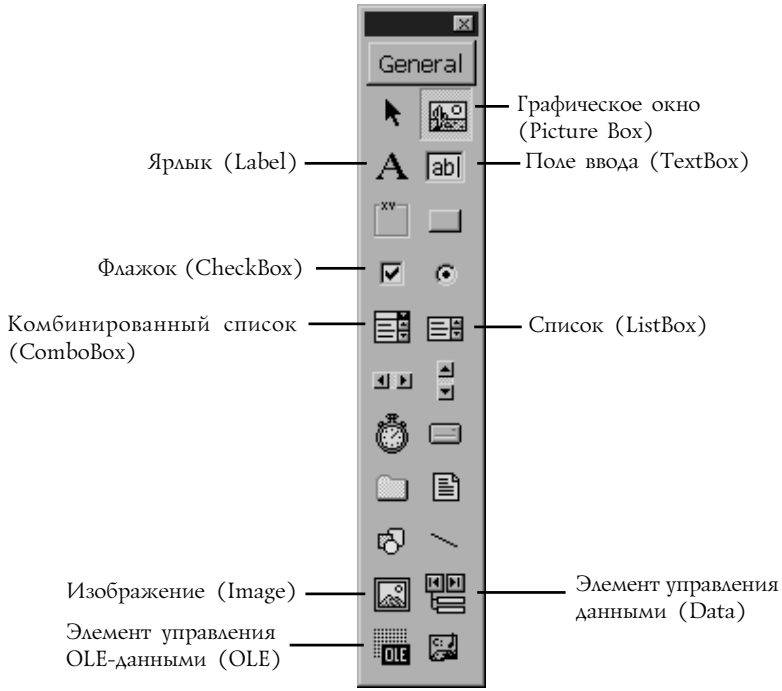


Рис. 3.3. Элемент управления данными и связанные элементы

Рис. 3.4. Форма "Каталог изданий" с элементом управления данными

Когда с помощью элемента **Data** пользователь приложения перемещается от одной записи к другой, связанный элемент отображает или даже позволяет редактировать (изменять) данные, содержащиеся в заданном для связанного элемента поле текущей записи. Кроме того, элемент **Data** позволяет создавать новые записи в базе данных, которые можно заполнять информацией с помощью связанных элементов.

Рассмотрим работу элемента **Data** и связанных элементов на примере уже упомянутой базы данных **Biblio.mdb**. Создадим форму с заголовком "Каталог издания", на которой разместим элемент **Data** со стандартным именем и заголовком **Data1** и 8 связанных элементов — полей ввода с именами **Text1—Text8** (без надписей); для каждого поля ввода с помощью ярлыков **Label1—Label8** зададим наименование отображаемой ими информации. На *рис.* 3.4 приведен вид созданной формы с указанием функций кнопок элемента управления **Data**.

Займемся настройкой элемента **Data** (с именем **Data1**) и полей ввода на базу данных.

❐ Для взаимодействия элемента **Data** и БД нужно определить по меньшей мере два свойства: **DatabaseName** (имя БД), значением которого является имя файла (с полным путем), содержащего БД, и **RecordSource** (источник записей). Значением свойства **RecordSource** может быть имя таблицы, входящей в состав БД. Но им может быть также имя запроса (**Query**), сформированного и сохраненного в самой БД, а также непосредственно текст инструкции-запроса.

Запрос — это инструкция, написанная на специальном языке SQL (*Structured Query Language*), результатом выполнения которой является набор записей (Set of records, или, как называется соответствующий объект в системе VB5, RecordSet).

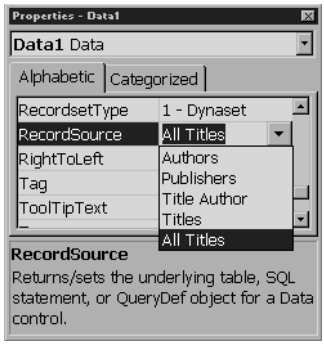
Вообще говоря, в VB5 существует 4 типа наборов записей: *таблицы* (***tables***) — физическая структура, содержащая реальные данные; *динамические наборы* (***dynasets***) — набор ссылок, обеспечивающих доступ к полям и записям в одной или нескольких таблицах; еще два типа — *снимки* (***snapshorts***), доступные только для чтения копии данных из одной или нескольких таблиц, с которыми мы не будем иметь дела.

Так как таблица — тоже набор записей, то, в общем, можно сказать, что элемент **Data** управляет набором записей, сопоставляемых ему через свойство **RecordSource**. Поля, из которых берутся данные для записей такого набора, могут находиться в разных таблицах БД.

Условием формирования записей набора из записей разных таблиц является равенство значений ключевых полей в этих записях. Например, из приведенной схемы БД **Biblio.mdb** видно, что мы можем формировать набор данных, содержащий имя автора (поле **Author** таблицы **Authors**) и название книги (поле **Title** таблицы **Titles**), объединяя данные из таких записей таблиц **Author** и **Titles**, в которых значения поля **Au_ID** из **Author** и значения поля **ISBN** из **Titles** находятся в единых записях таблицы **Title Author** (в ее соответствующих полях **Au_ID** и **ISBN**).

При создании базы данных формируемые запросы могут сохраняться в ней под заданными именами. Как мы сейчас увидим, есть такой запрос и в БД **Biblio.mdb**.

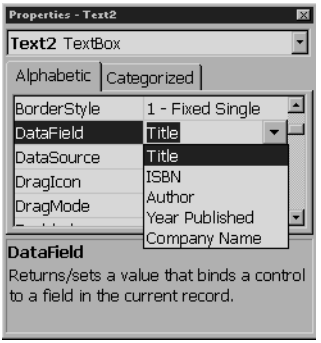
В окне свойств выберем свойство **DatabaseName**, щелкнув по нему мышью, а затем, щелкнув по появившемуся многоточию (знак вызова диалога), получим стандартный диалог для открытия файла. Откроем файл **Biblio.mdb**, путь в папку к которому — **c:\Program Files\DevStudio\Vb**. Затем, щелкнув по свойству **RecordSource**, получим значок для открывания списка (▼), щелкнув по которому откроем выпадающий список:



В данном списке — известные нам имена таблиц БД, а также имя запроса **All Titles**. Выберем запрос. Какие поля в него включены, мы увидим, настраивая любой из связанных элементов.

- Для настройки любого связанного элемента нужно определить минимум два его свойства: **DataSource** (источник данных) и **DataField** (поле данных). При этом в качестве *источника* данных указывается имя какого-либо элемента **Data**, а в качестве *поля данных* — имя поля в наборе записей, определяемом свойством **RecordSource** этого элемента **Data**.

Выделим на нашей форме поле ввода **Text2** (против надписи **"Наименование:"**) и в окне свойств установим (выбором из падающего списка) для свойства **DataSource** единственное в списке значение **Data1** (имя установленного элемента **Data**), а для свойства **DataField** таким же образом выбираем имя поля данных **Title**:

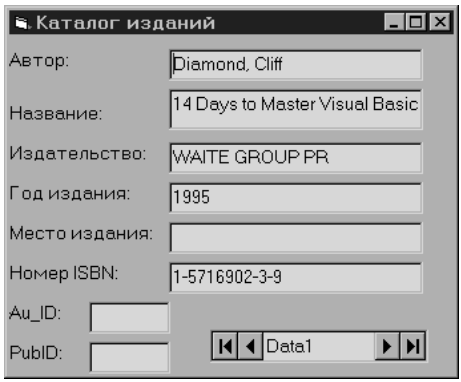


При этом мы можем узнать имена и остальных полей в сопоставленном элементу **Data** наборе записей **All Titles**.

- Последовательно выделяя мышью поля ввода против надписей **"Номер ISBN:"**, **"Автор:"**, **"Год издания:"** и **"Издательство:"**, установим таким же образом значения их свойств **DataSource** (**Data1**) и **DataField** (**ISBN, Author, Year Published** и **Company Name** соответственно). Элемент **Data1** и связанные с ним поля ввода готовы к работе!

Запустив приложение, мы посредством управляющих кнопок элемента **Data** "листаем" записи сопоставленного ему набора записей **All Titles**. Поля ввода допускают и редактирование данных в соответствующих полях базы данных, если только свойству **ReadOnly** элемента **Data** не установлено значение **True**. Кроме того, тип созданного набора записей должен допускать такую возможность; по умолчанию этот тип, задаваемый свойством **RecordsetType**, — **Dynaset** (Динамический) — эту возможность допускает. Удалять же и добавлять записи в базу данных с помощью элемента **Data** мы не можем: для этого нужно установить соответствующие командные кнопки и написать программный код. При написании программного кода, если такое случится, мы будем иметь дело с объектом **RecordSet** (Набор записей), доступ к которому можно осуществить посредством обращения к свойству **RecordSet**

элемента **Data**: **Data1.RecordSet**, например. Этот объект имеет ряд свойств и методов, обеспечивающих функции перемещения по записям, поиска и модификации записей. Мы же до сих пор не написали ни одной строки! И тем не менее можем спокойно листать и модифицировать данные в полях нашей БД:



Рассмотрим наконец самую "продвинутую" возможность работы с элементом **Data** (без написания кода): возможность задания управляемого набора записей не выбором имеющихся таблиц или хранимых запросов БД, а с помощью инструкции **SELECT** языка SQL, размещаемой в поле ввода свойства **RecordSource** (окно **Properties** проекта) элемента **Data**. Общая структура этой инструкции для такой задачи упрощенно следующая:

SELECT *Список_полей* FROM *Список_таблиц [, Отношения между_ними]*

(квадратные скобки указывают на необязательность названного внутри них элемента инструкции — здесь и далее).

А теперь запишем конкретный вариант такой инструкции для вывода всех 8 полей, отображение которых предусмотрено нашей формой "Каталог изданий":

```
SELECT a.Author, a.Au_ID, b.Title, b.[Year Published],
         b.ISBN, b.PubId, d.[Company Name], d.City
FROM Authors As a, Titles As b, [Title Author] As c,
      Publishers As d,
a INNER JOIN c ON a.Au_ID = c.Au_ID,
b INNER JOIN c ON b.ISBN = c.ISBN,
b INNER JOIN d ON b.PubId = d.PubId
```

Дадим некоторый комментарий.

- Вся инструкция записывается в поле свойства **RecordSource** в одну строку (по-другому и не получится).
- В списке таблиц для каждого элемента списка — имени таблицы — определен (для более краткой записи обращения к ней из других элементов инструкции) и ее *псевдоним* (после ключевого слова **As**). Введение псевдонимов необязательно.
- Имена таблиц или полей, содержащие внутри себя пробел, записываются в квадратных скобках.
- Если одно и то же имя поля имеется в различных таблицах из списка полей, то для уточнения объекта используется синтаксис составного имени: ***Имя_таблицы. Имя_поля*** (в качестве имени таблицы возможен псевдоним). Для большей ясности в приведенной инструкции все имена полей составные.
- Отношение между таблицами **a** и **c** задается в нашей инструкции фразой

```
a INNER JOIN c ON a.Au_ID = c.Au_ID
```

и означает следующее: "В формировании искомого набора записей участвуют только те записи таблиц **a** и **c**, у которых в одноименных полях **Au_ID** стоят одинаковые значения".

Настроив после ввода инструкции **SELECT** связанные элементы на поля сформированного набора записей, запустим приложение и убедимся в его работоспособности (*рис.* 3.5).

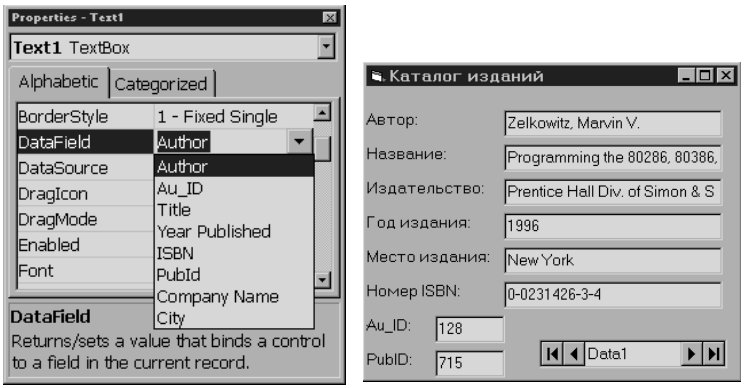


Рис. 3.5. Список полей набора записей, созданного по инструкции SQL, введенной как значение свойства **RecordSource** в окне **Properties**, и просмотр записей этого набора посредством элемента **Data** в связанных элементах

Отображение записей БД в списках

Помимо указанных стандартных элементов управления, используемых в функции связанных, существует 3 специализированных связанных элемента для списочно-табличного представления содержимого БД. Эти элементы, как и другие компоненты **ActiveX**, перечисляются в диалоге **Components** (Компоненты), который мы уже вызывали для использования элемента **Мультимедиа** в "Словарной обучалке".

Вызовем диалог **Components**, во вкладке **Controls** установим флажки напротив названий интересующих нас компонентов **Microsoft Data Bound Grid Control** и **Microsoft Data Bound List Controls 5.0** и щелкнем по кнопкам **Применить** и **OK**. В палитре инструментов появляются 3 новых компонента (*рис.* 3.6).

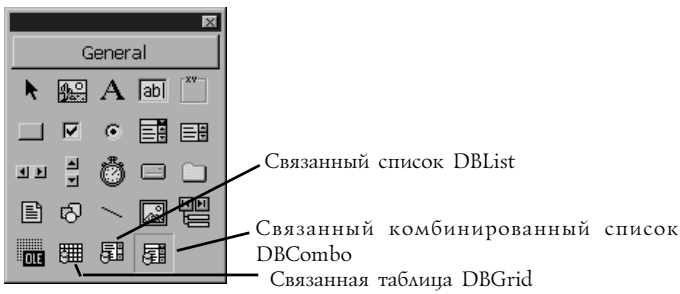


Рис. 3.6. Специализированные связанные элементы

Элементы **DBList** и **DBCombo** позволяют автоматически отображать информацию из набора записей в виде списка, в то время как соответствующие стандартные элементы **ListBox** и **ComboBox** могут это делать только через применение метода **AddItem** (этот метод к тому же не действует, если эти списки связаны с элементом **Data**). Но, помимо просто отображения данных полей БД, эти элементы дают возможность создания связей (путем копирования элементов одной таблицы — например, идентификационных значений — в другую таблицу) между какими-либо полями различных таблиц БД.

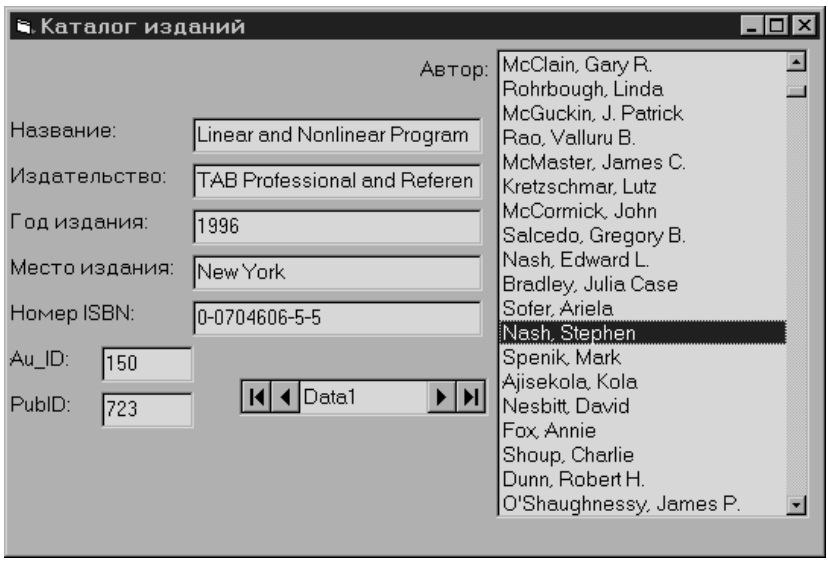
Для использования элементов **DBList** и **DBCombo** нужно определить как минимум 5 их свойств:

- **RowSource** — имя элемента **Data** — источника записей для списка.
- **ListField** — имя поля, данные которого отображает список.

- **BoundColumn** — имя поля, содержащего идентификационные значения для данных поля **ListField**, отображаемого списком; при выборе из списка соответствующее идентификационное значение копируется в поле **DataField**; может совпадать с **ListField**, если дублирование информации не смущает разработчика.
- **DataSource** — имя элемента **Data**, связанного с целевым набором записей, формируемого в процессе выбора из списка; может совпадать с **RowSource**.
- **DataField** — имя целевого поля в целевом наборе записей, куда копируются идентификационные значения поля **BoundColumn** для выбираемых из списка элементов (либо сами эти элементы, если поле **BoundColumn** совпадает с полем **ListField**); тип данного поля должен совпадать с типом поля **BoundColumn**.

Если мы хотим просто отображать в виде списка какое-либо поле некоторого набора записей, то **RowSource** и **DataSource** имеют своими значениями один и тот же элемент **Data**. В этом случае, естественно, значения для свойств **BoundColumn** и **DataField** не указываются.

Вот как выглядит окно приложения "Каталог изданий", в котором поле авторов из сформированного по инструкции **Select** указанного набора записей отображается с помощью списка **DBList**:



При этом мы можем перемещаться и производить выбор пунктов списка авторов с помощью мыши и полосы прокрутки совершенно независимо от состояния элемента **Data**; в свою очередь, все свойства управления набором с помощью элемента **Data** сохраняются. Модифицировать пункты списков, используя стандартные клавиатурные возможности (как в связанных полях ввода), невозможно даже с помощью поля ввода элемента **DBCombo**.

Информационная же таблица **DBGrid** позволяет просто просматривать одновременно в табличном виде весь набор записей, являющийся источником записей (**RecordSource**) какого-либо элемента **Data**. Для этого имя такого элемента надо указать значением свойства **DataSource** таблицы **DBGrid**. Используя мышь и линейки прокрутки, мы можем перемещаться в любое место таблицы (независимо от состояния элемента **Data**) и выбирать в ней нужную запись. Вот как выглядит окно приложения, в котором таблица позволяет просматривать полностью все тот же набор записей:

Author	Au_ID	Title	Year	ISBN	PubId	Company Nam
Kretzschmar, Lutz	12963	Create Your Own Mu	1994	0-0704603-4-5	724	Windcrest Book
McCormick, John	12968	Create Your Own Mu	1994	0-0704603-4-5	724	Windcrest Book
Salcedo, Gregory B.	3389	Database Marketin(	1993	0-0704606-3-9	723	TAB Profession
Nash, Edward L.	3991	Database Marketin(	1993	0-0704606-3-9	723	TAB Profession
Bradley, Julia Case	146	Linear and Nonlinea	1996	0-0704606-5-5	723	TAB Profession
Sofer, Ariela	149	Linear and Nonlinea	1996	0-0704606-5-5	723	TAB Profession
Nash, Stephen	150	Linear and Nonlinea	1996	0-0704606-5-5	723	TAB Profession
Spenik, Mark	7059	Sybase/Sql Server	1995	0-0704629-9-2	63	MCGRAW HILL
Ajisekola, Kola	7063	Sybase/Sql Server	1995	0-0704629-9-2	63	MCGRAW HILL
Nesbitt, David	4289	Sybase/Sql Server	1995	0-0704629-9-2	63	MCGRAW HILL
Fox, Annie	11450	Armchair Basic : An	1983	0-0704785-8-9	79	OSBORNE MC(
Shoup, Charlie	11396	Armchair Basic : An	1983	0-0704785-8-9	79	OSBORNE MC(
Dunn, Robert H.	13424	Invest Like the Best	1994	0-0704798-4-4	63	MCGRAW HILL
O'Shaughnessy, James P.	13430	Invest Like the Best	1994	0-0704798-4-4	63	MCGRAW HILL

Теперь мы вполне достаточно знаем о том, как работать с БД при помощи элемента **Data** и связанных элементов управления (и при этом без всякого программирования). Мы, правда, еще не научились добавлять и удалять записи в БД с помощью этих средств. Но главное, к чему нам давно уже пора приступить, — это создание новой БД средствами VB5.

Создание файла БД и формы для его заполнения

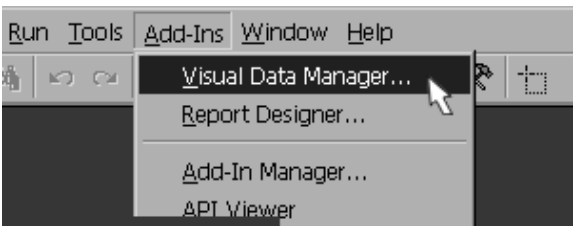
Вернемся снова к нашему проекту "Словарной обучалки". В первоначальном проекте мы предполагали хранить данные со словарными, изобразительными и звуковыми отображениями предметов в двумерной БД, состоящей из единственной таблицы (назовем ее *Subject* — Предмет):

RusWord	EnglWord	Image	Wave
DOG	Собака	<ссылка на файл>	<ссылка на файл>
DOLPHIN	Дельфин	<ссылка на файл>	<ссылка на файл>
DONKEY	Осел	<ссылка на файл>	<ссылка на файл>

Впоследствии мы нашли способ решения нашей задачи и без создания такой БД, на основе списков. Сейчас же мы вернемся к первоначальному замыслу и будем учиться создавать БД и заполнять ее средствами VB5.

Откроем новый стандартный проект и выполним следующую последовательность действий сначала по созданию файла БД в формате Access, а затем по созданию формы для заполнения БД данными.

- Выбираем команду **Visual Data Manager** из пункта **Add-Ins** (Надстройки) главного меню:



Появляется окно **VisData**, в верхней части которого строка меню и панель инструментов. В пункте меню **File** содержится набор команд для начала работы по созданию БД.

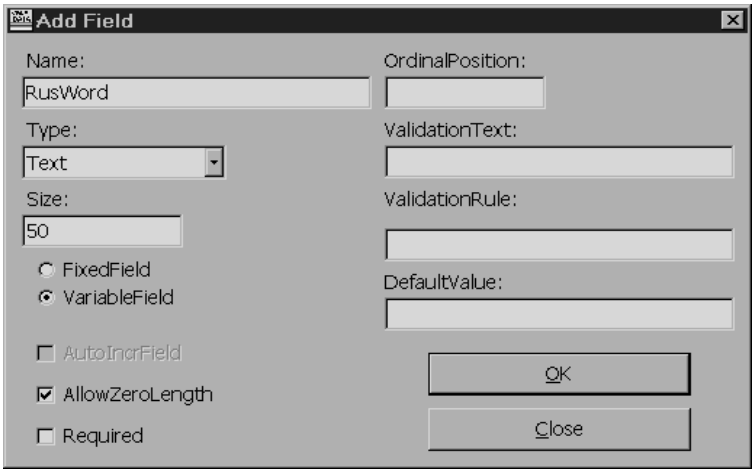
- В пункте **File** выбираем формат будущей БД командой **New... ➢ ➢ Microsoft Access ➢ Version 7.0 MDB** с помощью последовательности выпадающих списков. Появляется диалог (используемый в VB5 для сохранения файлов) с именем **Select Microsoft Access**

Database to Create, в котором мы выбираем или создаем каталог, где предполагаем сохранить файл с нашей БД, и задаем имя этого файла **(obuchalka.mdb)**.

- После сохранения в каталоге имени будущей БД это же имя появляется в заголовке окна **VisData**, а внутри него появляются 2 новых окна: **SQL Statement** (Инструкции SQL), которое нам не понадобится, и **Database Window** (Окно базы данных). Щелкаем в нем правой клавишей и из появившегося контекстного меню выбираем **New Table** (Новая таблица):

— после чего на экране появляется диалог **Table Structure** (Структура таблицы), используемый для задания структуры и свойств будущей таблицы.

- В поле **Table Name** (Имя таблицы) диалога введем имя создаваемой таблицы **Subject**, после чего щелкнем по кнопке **Add Field** (Добавить поле), расположенной под окном списка **Field List.** Появляется диалог **Add Field**:

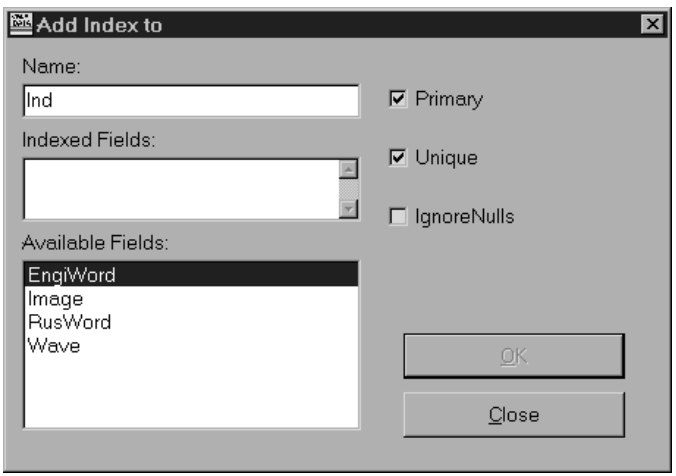


- В диалоге **Add Field** мы будем последовательно вводить характеристики каждого из полей таблицы, заканчивая их ввод щелчком по кнопке **OK**. После ввода данных очередного поля его имя появляется в списке **List** диалога **Table Structure**.

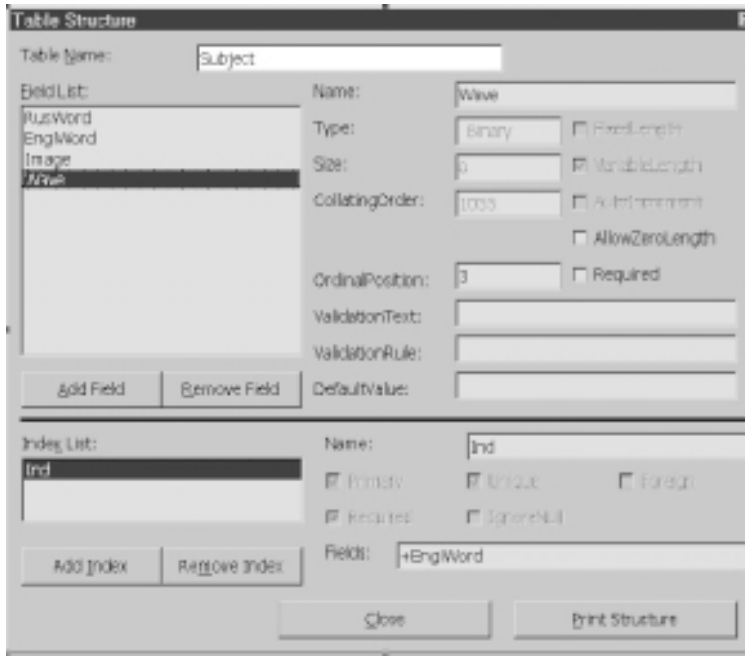
  Для полей **RusWord** и **EnglWord** достаточно ввести только их имена в поле ввода **Name**, а значения по умолчанию в полях **Type** (Тип) и **Size** (Размер) оставить неизменными (**Text** и **50** соответственно). Это означает, что в данных полях должны располагаться значения строкового типа, причем длина строки не должна превышать (включена радиокнопка **VariableField** — Переменное поле) 50 символов. Для полей **Image** и **Wave**, кроме их имен, в поле **Type** вводится их тип **Binary** (Двоичный).

  По окончании ввода данных по всем полям диалог **Add Field** закрывается кнопкой **Close** (Закрыть).

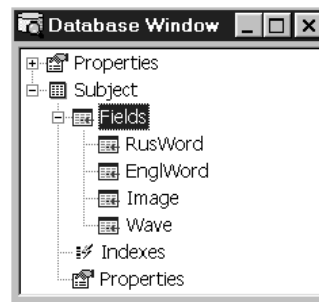
- Для поля, по которому предусмотрена сортировка (а также поиск), можно добавить *индекс* — специальную таблицу, которая будет содержать, во-первых, отсортированные значения индексируемого поля, а во-вторых, для каждого такого значения — физический номер содержащей его записи данной таблицы БД (для одинаковых индексированных значений таблица индексов расположит их в очередности соответствующих физических номеров записей данной таблицы БД). Щелкаем по кнопке **Add Index** (Добавить индекс), расположенной под окном списка **Index List** формы **Table Structure**. В появившемся диалоге **Add Index to** мы задаем имя индекса (**Ind**) и выбираем в окне списка **Available Fields** (Доступные поля) поле **EnglWord**:



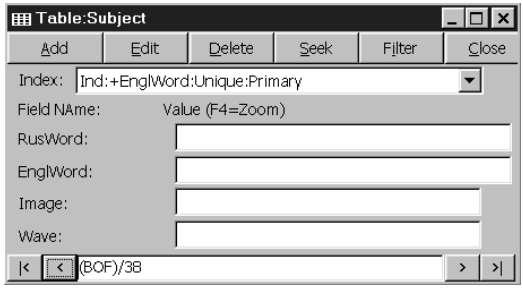
— после чего диалог закрываем. Имя индекса появляется в соответствующем списке формы **Table Structure**, заполненной нужным нам образом:



- Выполняем сохранение созданной структуры таблицы: щелкаем по кнопке **Build the Table** (Построить таблицу). Окно **Table Structure** закрывается, а в окне **Database Window** на дереве структуры и полей создаваемой БД появляется узел таблицы **Subject**, развернув который получим следующий вид структуры БД:

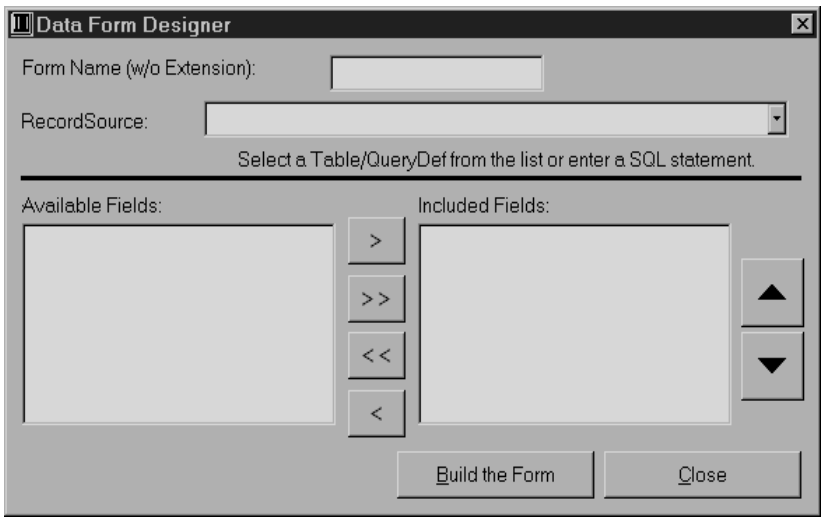


Сделав далее двойной клик по узлу с именем таблицы **Subject**, мы получим форму с именем **Table:Subject** для ввода данных в созданную таблицу:

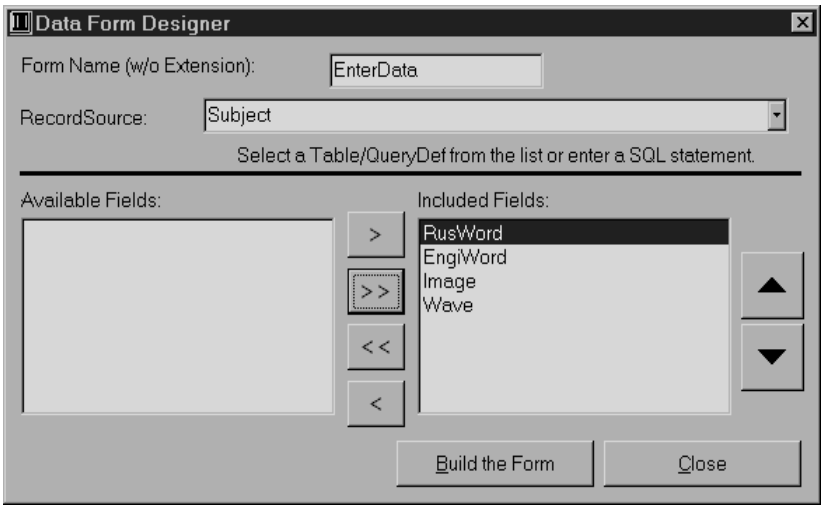


Однако эта форма непригодна для реальной работы (не поддерживает ввод русскоязычного текста) и, кроме того, не может быть использована независимо от надстройки. Поэтому мы закроем ее кнопкой **Close** и займемся созданием *независимой* формы для ввода данных в нашу БД.

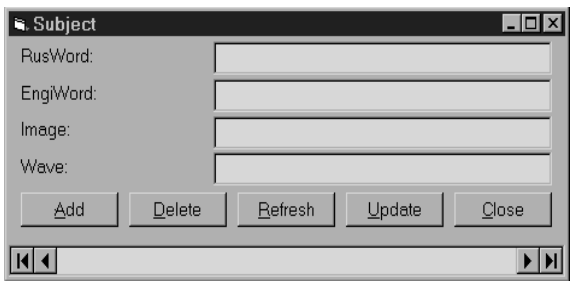
- Для создания независимой формы для ввода, удаления и модификации записей в таблицу **Subject** войдем в пункт **Utility** (Утилиты) окна надстройки **VisData** и выберем **Data Form Designer** (Конструктор формы БД). Появляется следующий диалог:



- Вводим имя создаваемой формы (**frmEnter_DB**, например) и из списка **RecordSource** (Источник записи) выбираем таблицу, для ввода в которую создается форма (таблица **Subject**). В списке **Available Fields** в левой части диалога появляются имена полей таблицы. Щелкаем по кнопке >> для передачи всех имен в правый список **Included Fields** (Включенные поля) и получаем:



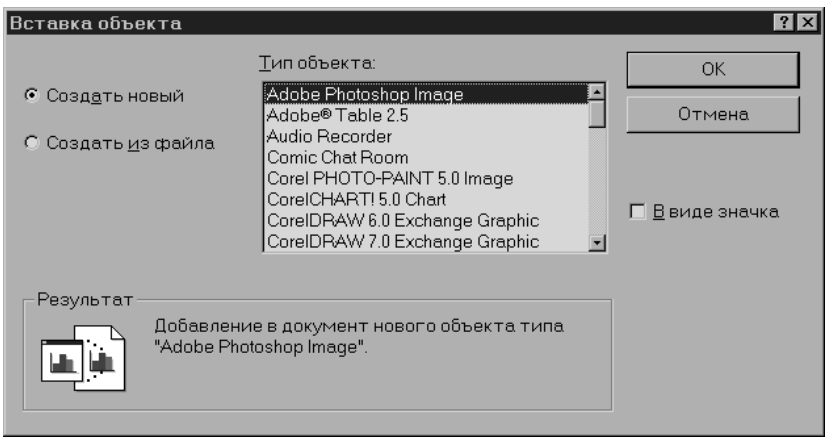
Щелкаем по кнопке **Build the Form** (Создать форму) и получаем требуемую экранную форму для ввода данных в нашу БД:



Данную форму мы можем включать в любой проект, вызвав в нем диалог **Add Form** со вкладкой **Existing** (Существующий), — щелчком мыши по пиктограмме **Add Form**, дублирующей соответствующую команду из пункта главного меню **Project**:

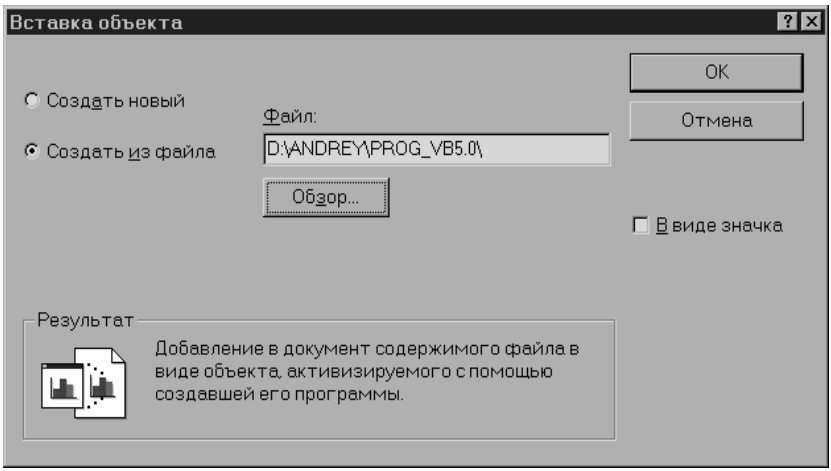
В данный момент, однако, созданная форма уже включена в текущий вновь открытый стандартный проект, в котором с самого начала уже имелась форма **Form1**. Если мы, захотев начать ввод данных в БД, запустим наш проект на выполнение, то загрузится именно форма **Form1**. Для загрузки же формы **frmEnter_DB** нам придется вызывать метод **Show**, помещая вызов **frmEnter_DB.Show**, например, в обработчик события **Load** формы **Form1**. Но в нашем случае можно поступить и иначе. Выполним команду **Properties** пункта главного меню **Project** и в окне появившегося диалога укажем форму **frmEnter_DB** как **StartUp Object** (Стартовый объект). Теперь именно эта форма будет загружена после запуска проекта на выполнение.

Запустим проект и начнем вводить данные в поля данных формы **frmEnter_DB**. Ввод новых записей в таблицу БД начинается с того, что мы при необходимости перемещаемся на ее последнюю запись (для пустой таблицы этого делать не нужно) и щелкаем по кнопке **Add**. При этом имевшиеся в полях данные пропадают. Текстовые данные вводятся далее обычным образом. Для ввода же в поле OLE-данного необходимо произвести двойной клик в этом поле, после чего появляется следующий диалог:



Включена по умолчанию кнопка **Создать новый**, и задан список разновидностей вставляемых OLE-объектов, которые можно создать с помощью соответствующих приложений. Для того чтобы создаваемые картинки открывались в OLE-объекте, необходимо создавать их в формате BMP или PCX, что можно сделать в приложении Corel PhotoPaint. Для создания звуковых файлов выбирается тип объекта **Звукозапись**, которому соответствует приложение **Фонограф**. Создание графического объекта с помощью приложения не означает, разумеется, что оно обязано быть там "нарисовано"; нужная картинка может быть просто открыта в приложении.

Если же у нас уже имеются заготовленные картинки в файлах указанных форматов или звуковые файлы, то нужно в диалоге **Вставка объекта** выбрать опцию **Создать из файла**:



— после чего посредством файлового броузера открыть нужный файл в поле OLE-объекта.

- После того как все данные будут введены в поля формы, необходимо щелкнуть по кнопке **Update** (Обновить). Данные запишутся в файл БД, и отображаемый в поле элемента **Data** номер текущей записи увеличится на 1.

Сходным образом можно произвести и модификацию любой записи таблицы. Удалить запись можно кнопкой **Delete**. По окончании корректировки БД форма закрывается кнопкой **Close**.

Работа с базой данных "Словарной обучалки"

В нашем проекте уже создана форма (со свойством **Name= frmEnter_DB**), которая позволит самому пользователю "обучалки" дополнять и модифицировать предметный состав обучалки. Осталось дело за малым — использовать созданную по умолчанию пустую форму **Form1** для задания основного окна приложения.

В окне работающей новой версии нашей "обучалки" должны быть видны:

- окно OLE-объекта (OLE1) с картинкой;
- поле ввода для вывода русского слова **(txtRus)**;
- комбинированный список английских слов-переводов **(dbcEngl)**;
- кнопки запуска режима тестирования **(cmdStart)**, прерывания тестирования **(cmdBreak)**, запуска режима корректировки БД **(cmdAdd)** и выхода из программы **(cmdEnd)**.

Примерное расположение указанных элементов на форме (а также установленные во время проектирования значения свойства **Caption** для тех из них, у кого это свойство есть) может быть таким:

Кроме указанных видимых в фазе выполнения элементов, на форме произвольным образом располагаются следующие невидимые элементы (те, которые вообще не имеют свойства **Visible**, а также те, у которых оно при проектировании установлено в **False**):

- элемент **Data (Data1);**
- OLE-объекты **OLE1** (для звукового отображения предметов) и **OLE2** (для отображения звукового сигнала);
- таймеры **tmrSec, tmrDelay** и **tmrTest** того же назначения и свойств, что и в предыдущей версии;
- массив ярлыков **(lblFault)**, посимвольно отображающих надпись "Неверно!" в вертикальном направлении в границах элемента **OLE1**;
- ярлык **(lblEngl)**, связанный с элементом **Data1** по полю английских слов-переводов **EnglWord**.

Принцип работы проектируемой программы подобен уже рассмотренному в предыдущем варианте. Единственной особенностью является то, что сравниваются не ключевые значения записей в разных списках, а сами значения поля **EnglWord** — для записи, случайно выбранной программой (отображается в связанном ярлыке **lblEngl**), и для пункта, выбранного обучаемым в комбинированном списке **dbcEngl** с помощью мыши либо путем ввода с клавиатуры в поле ввода данного списка.

Перед началом установки элементов и их свойств необходимо скопировать файл базы данных **obuchalka.mdb** в ту директорию, где расположены файлы программы, входящие в проект. Обратим внимание на некоторые свойства устанавливаемых элементов.

Свойству **DatabaseName** элемента **Data1** при проектировании задается текущий путь к файлу БД **obuchalka.mdb**. Однако мы обязаны предусмотреть, что фаза выполнения может происходить необязательно на том же компьютере, что и проектирование, а потому путь **(Path)** по файловой системе к файлу БД может изменяться. К счастью, есть возможность программным образом переназначить этот путь так, что он будет определяться относительно каталога, в котором размещен либо файл проекта (при запуске интерпретатора кода программы из системы VB5), либо EXE-файл приложения (при независимом запуске загрузочного модуля).

Для формирования набора записей **RecordSet**, в котором записи будут отсортированы в лексикографическом порядке по полю **EnglWord**, в качестве значения свойства **RecordSource** элемента **Data1** запишем следующую инструкцию **SELECT**:

```
SELECT * FROM Subject ORDER BY EnglWord
```

Здесь **"*"** означает "все записи", а **ORDER BY** — сортировку (по умолчанию — в порядке возрастания, т.е. "по алфавиту").

У связанных элементов — **OLE1, OLE2, txtRus, lblEngl** — установим для свойства **DataSource** значение **Data1**, а для свойства **DataField** — имя отображаемого поля набора записей (у элемента **dbcEngl** эти значения устанавливаются у свойств **RowSource** и **ListField** соответственно).

У элемента **OLE3** в свойстве **SourceDoc** посредством открывающегося диалога **Вставка объекта** после установки опции **Создать из файла** находится средствами броузера файл с нужным нам звуковым объектом (звуковым сигналом), который и вставляется в OLE-объект.

Установка значений других свойств элементов формы (и ее самой) уже нам знакома и не представляет никаких сложностей.

Перейдем к рассмотрению кода, реализующего программный алгоритм.

❐ Возможно, вы уже обратили внимание на отдельные повторяющиеся фрагменты в разных обработчиках предыдущего варианта программы. Для уменьшения объема кода и ускорения компиляции такие фрагменты можно, как мы уже говорили, оформить в виде независимых, расположенных на одном с обработчиками уровне, процедур-подпрограмм. Синтаксически они ничем не отличаются от обработчиков и могут просто вводиться с клавиатуры. Но можно воспользоваться и командой **Add Procedure** пункта **Tools** главного меню. В появившемся диалоге вводим имя создаваемой подпрограммы и переключаем опцию **Public** (по умолчанию) на **Private** (не нужно без надобности делать общедоступными имена из файла проекта). После выбора **OK** в тексте кода появляется заготовка создаваемой подпрограммы, аналогичная заготовкам обработчиков. Приводим секцию глобальных объявлений и тексты вспомогательных подпрограмм:

```
Dim li As Integer, ii_o As Integer, t As Integer, _
    n As Integer, nf As Integer, rc As Integer
' rc - количество записей в БД
Public Sub NewSelect() 'Случайный выбор очередного предмета
 li = Int(rc * Rnd)
 If li = li_o Then
  If li = rc - 1 Then
   li = li - 1
  Else
   li = li + 1
  End If
 End If
 Data1.Recordset.AbsolutePosition = li
 li_o = li
End Sub
```

```
Private Sub CheckChoice() 'Проверка ответа обучаемого
 n = n + 1
 If lblEngl.Caption = dbcEngl.Text Then
  OLE2.DoVerb
 Else
  Call OutFault(True)
  OLE3.DoVerb
  nf = nf + 1
 End If
 tmrDelay.Enabled = True
End Sub

Private Sub OutFault(boo As Boolean) 'Управление надписью
 Dim j As Integer                     ' "Неверно!"
 For j = 0 To 6
  lblFault(j).Visible = boo
 Next
End Sub
```

Отметим использование следующих *новых* программных элементов.

К ним относится упоминавшийся ранее объект **RecordSet**, к которому мы обращаемся через одноименное свойство элемента **Data1**. Свойство **AbsolutePosition** этого объекта возвращает или устанавливает номер текущей записи в нем (от **0** до **rc—1**, где **rc** — число записей в **RecordSet**). (Значение переменной **rc**, как мы увидим, устанавливается ранее в обработчике загрузки основной формы с помощью свойства **RecordCount**, возвращающего полное число записей в **RecordSet**. Причем для определения значения этого свойства необходимо вначале "пролистать" все записи, перейдя к последней с помощью метода **MoveLast.**)

Другим новым элементом кода является метод **DoVerb** OLE-объектов, производящий открытие OLE-объекта (для редактирования или иных действий по умолчанию). Открытие "звукового" OLE-объекта приводит к проигрыванию его содержимого.

- В обработчике загрузки основной формы мы теперь должны инициализировать объект **RecordSet** и определять число записей в нем. Инициализация (то есть открытие доступа к нему в программе) объекта **RecordSet**, порожденного элементом **Data** с заданным при проектировании источником данных, может происходить автоматически — при первоначальном отображении значения поля связанным с элементом **Data** элементом. Но если источник данных устанавливается программно, то объект **RecordSet** должен инициализироваться методом **Refresh**. Как же устанавливается источник данных, если мы, как уже говорилось, хотим, чтобы его файл указывался *относительно того каталога, в котором расположено*

приложение? Для этого используется объект **App**, который определяет или уточняет информацию о названии приложения, о пути к его исполняемому файлу и файлам справки, о присутствии выполняющихся экземпляров этого приложения и т.д.:

```
Private Sub Form_Load() 'Инициализация набора записей
 Data1.DatabaseName = App.Path & "/obuchalka.mdb"
 Data1.Refresh
 Data1.Recordset.MoveLast
 rc = Data1.Recordset.RecordCount
 Randomize
 Call NewSelect
End Sub
```

После инициализации объекта **RecordSet** число записей в наборе определяется, как мы видим, с помощью метода **MoveLast** и свойства **RecordCount** этого объекта.

- Проверка слова-перевода, выбранного обучаемым, осуществляется в двух нижеприведенных обработчиках: события **Click** и события **KeyPress** для списка **dbcEngl**. В последнем случае проверка производится, разумеется, не для каждого нажатия на клавишу **(KeyPress)** при вводе слова-перевода, а только при завершающем ввод нажатии клавиши Enter. Это оказывается возможным потому, что при вызове обработчика этого события ему через аргумент передается ASCII-код нажатой клавиши (код клавиши Enter — 13). Значение соответствующего параметра обработчика **KeyAscii** анализируется на равенство этому коду:

```
Private Sub dbcEngl_Click(Area As Integer) 'Клик слова-
 Call CheckChoice                          'перевода в списке
End Sub

Private Sub dbcEngl_KeyPress(KeyAscii As Integer) 'Ввод
 If KeyAscii = 13 Then 'слова-перевода в поле списка
Call CheckChoice
 End If
End Sub
```

- Для двух новых кнопок — корректировки БД и прерывания теста — появились два новых обработчика:

```
Private Sub cmdAdd_Click() 'Изменение состава БД
 Call cmdBreak_Click
 frmEnter_DB.Show
End Sub
```

```
Private Sub cmdBreak_Click() 'Прерывание теста
 cmdStart.Caption = "Запуск теста (60 сек.)"
 tmrDelay.Enabled = False
 tmrSec.Enabled = False
 Call OutFault(False)
 tmrTest.Enabled = False
 Call NewSelect
 dbcEngl.SetFocus
End Sub
```

Использованный для вызова формы метод **Show** (Показать), во-первых, загружает форму, если она не была загружена, а во-вторых, делает ее видимой. Снова скрывает форму (но не выгружает ее, т.е. программный доступ к ней сохраняется) метод **Hide** (Скрыть).

В первом обработчике второй вызывается так, как будто это обычная, реализующая вспомогательный алгоритм подпрограмма.

- Остальные обработчики нашего приложения принципиально ничем не отличаются от своих аналогов в предыдущем варианте:

```
Private Sub cmdStart_Click() 'Запуск теста
 t = 0
 n = 0
 nf = 0
 frmEnter_DB.Hide
 Call NewSelect
 dbcEngl.SetFocus
 tmrSec.Enabled = True
 tmrTest.Enabled = True
End Sub

Private Sub tmrDelay_Timer()
 Call NewSelect
 tmrDelay.Enabled = False
 Call OutFault(False)
End Sub

Private Sub tmrSec_Timer()
 t = t + 1
cmdStart.Caption = t & " сек"
End Sub

Private Sub tmrTest_Timer()
 Dim b As Integer
 cmdStart.Caption = "Запуск теста (60 сек)"
 tmrDelay.Enabled = False
 tmrSec.Enabled = False
 Call OutFault(False)
 Select Case n - nf
```

```
Case Is > 8
   If nf = 0 Then
    b = 5
   Else
    b = 4
   End If
   If nf = 0 Then
    b = 4
   Else
    b = 3
   End If
  Case 3 To 5
   If nf = 0 Then
    b = 3
   Else
    b = 2
   End If
  Case Else
   b = 2
 End Select
 MsgBox "Дано " & n & " ответов; число Ваших ошибок: " _
        & nf & Chr(13) & "Отметка: " & b, vbInformation
 tmrTest.Enabled = False
 Call NewSelect
End Sub
```

- Последнее, что мы сделаем, — это заменим в обработчике события **Click** для кнопки **Close** формы **frmEnter_DB** вызов метода **Unload Me** (при его выполнении курсор "заклинивает" на форме песочных часов), выгружающего эту форму по окончании работы с ней, на следующий код:

```
Hide
frmObuchalka.dbcEngl.ReFill 'обновления содержимого
     'связанного списка и его перерисовка
     'после корректировки БД
frmObuchalka.dbcEngl.SetFocus
```

Кроме того, чтобы фокус гарантированно передавался после загрузки формы **frmObuchalka** сразу на поле ввода комбинированного списка, создадим обработчик:

```
Private Sub txtRus_GotFocus()
 dbcEngl.SetFocus
End Sub
```

Работа с БД: о чем мы только упомянем

Средства работы с БД, некоторые из которых мы изучили в данной главе, представляют собой элементы программного интерфейса, позволяющего приложению *запрашивать* у базы данных тот или иной сервис. Для создания такого интерфейса в VB5 существуют, как уже упоминалось, "объекты доступа к данным" (DAO — *Data Access Objects*), одним из которых, в частности, является известный нам объект **Recordset**. Но база данных — это просто файл во внешней памяти; *запросы* же поступают к так называемому *ядру* базы данных, которая вместе с программным интерфейсом составляет СУБД. Ядром БД в системе VB5 является процессор базы данных Microsoft Jet Engine версии 3.5, называемый также просто Jet-машиной. Подобный же процессор используется, в частности, и в СУБД MS Access 7.0. Поэтому особенно легко работается средствами VB5 именно с БД этого формата. На самом деле в VB5 имеется возможность работы с БД всевозможных форматов, в том числе и с подключением их к Jet-машине.

Мы, однако, до сих пор ни словом не обмолвились о том, что, помимо программного интерфейса, рассчитанного на управление Jet-машиной, в VB5 существует реализация и программного интерфейса, рассчитанного на управление так называемыми "драйверами ODBC" (*Open Database Connectivity* — "открытые средства связи с базами данных"), представляющими динамические библиотеки функций подключения к БД разных типов. Если средства DAO и Jet-машины рассчитаны на БД, расположенные на персональном компьютере (Access, FoxPro, dBase, Paradox, Lotus), то средства драйверов ODBC предполагают работу в системе клиент—сервер с серверными БД (SQL Server, Oracle и др.), т.е. когда реальная обработка записей БД осуществляется СУБД, расположенной на файловом сервере, а клиентское приложение на удаленном компьютере только выдает запросы серверу (на языке SQL, например). Программный интерфейс с драйверами ODBC организуется уже не средствами DAO, а средствами "объектов удаленных данных" (RDO — *Remote Data Objects*).

О RDO мы больше ничего не скажем, кроме того, что эти средства доступны только в Enterprise-редакции VB5. А вот про DAO дадим некоторую наиболее общую информацию, которая должна помочь в последующей самостоятельной ориентации.

Прежде всего отметим, что элемент **Data**, который мы активно использовали, не является объектом DAO, хотя и может использоваться совместно с ними. Так мы и делали, работая с объектом DAO **Recordset**,

обращаясь к нему через свойство **Recordset** элемента **Data**. Достоинством элемента **Data** при работе с БД является простота программирования (особенно с учетом возможностей связанных элементов), а также возможность определения БД и наборов записей на этапе проектирования простым выбором в диалогах и списках. Нередко используется создание набора записей как объекта DAO с последующей передачей его элементу **Data** посредством инструкции **Set**.

Объекты DAO представляются либо как свойства (других объектов DAO, элемента **Data**), либо как *объектные переменные* (*типа* соответствующего *объекта* DAO). Вот пример создания объекта **Database** путем открытия БД методом **OpenDatabase** и последующее создание набора данных:

```
Dim dbName As Database, recName As Recordset
Set dbName = OpenDatabase("Путь к файлу БД")
Set recName = dbName.OpenRecordset("Имя таблицы, " _
              "запроса либо инструкция SELECT")
```

Созданный таким образом набор можно далее передать для управления и отображения в связанных элементах элементу **Data** (с именем **datName**):

```
Set datName.Recordset = recName
```

Преимущества такого подхода очевидны: мы можем формировать передаваемый элементу **Data** набор записей в фазе выполнения приложения по тому или иному запросу пользователя.

Объект **Recordset** имеет множество методов, позволяющих работать с наборами данных: добавлять **(AddNew)**, удалять **(Delete)**, редактировать поля **(Edit)**. Перед удалением или редактированием записи на нее необходимо предварительно позиционироваться с помощью методов группы **Move, Find** или **Seek** (контролируя свойство **NoMatch**, устанавливаемого в **True** при нерезультативном поиске), а также известного нам свойства **AbsolutePosition**. После обращения к методам **AddNew** или **Edit** поля вводимой или редактируемой записи заполняются новыми значениями, причем обращение к тому или иному полю производится указанием строки с его именем в круглых скобках после имени объекта **Recordset**. Окончательная физическая модификация БД производится последующим обращением к методу **Update**.

Вот пример редактирования поля **PhoneNumber** набора записей **Person** (находим запись со старым номером телефона и заменяем старый номер на новый):

```
Person.FindFirst "PhoneNumber = '" + gsOldNumber + "'"
'Параметром метода FindFirst является строковое выражение,
'изображающее операцию сравнения. Обратите внимание на
'ее специфический синтаксис
```

```
If Person.NoMatch Then
  MsgBox "Запись отсутствует"
Else
  Person.Edit
  Person("PhoneNumber") = gsNewNumber
  Person.Update
End If
```

Помимо модификации БД, объекты DAO позволяют создавать новые БД в процессе работы приложения (а также менять структуру БД). Это бывает нужно при коммерческой поставке приложения, когда громоздкую БД, с которой работает приложение, проще создать "по месту", чем включать в поставку.

Для создания новой БД используется метод объекта **Workspace** (рабочая область) **CreateDatabase** (создать БД). Объект **Workspace** определяет отдельный сеанс работы, отдельный поток обработки данных ("транзакцию") ядром БД (объект **DBEngine**). Работа с несколькими рабочими пространствами (они различаются с помощью механизма индексации) организуется в многопользовательских системах для различения пользователей. В обычных случаях, а также по умолчанию работа ведется в рабочей области **Workspace(0)**. Метод **CreateDatabase** создает пустой файл БД; после этого необходимо задать структуру БД, используя для представления ее таблиц объект **TableDef**, для представления полей в таблицах — объект **Field**. Вот как выглядит процесс создания новой БД средствами DAO (для простоты в БД одна таблица, в которой одно поле):

```
Dim dbName As Database, tblName As TableDef, fldF1 As Field
Set dbName = WorkSpace(0).CreateDatabase("Путь_к_файлу_БД", _
                                         Порядок)
Set tblName1 = dbName.CreateTableDef("Таблица_1")
Set fldF1 = tblName1.CreateField()
fldF1.Name = "Поле_1"
fldF1.Type = Тип_поля 'Задается именованной константой
fldF1.Size = Размер_поля '(Существует не для всякого типа поля)
tblName1.Fields.Append fldF1 'Добавление описанного поля
                             'в таблицу методом Append
```

Если с типами и размерами полей мы уже познакомились при создании БД **obuchalka.mdb,** то параметр ***Порядок*** метода **CreateDatabase** нам еще не знаком. Он определяет (заданием именованной константы) язык (кодовую страницу), по алфавиту которого будут браться символы при сравнении строк. Значение **dbLangGeneral** задается для использования англо-американской кодовой страницы; для сравнения в соответствии с русским алфавитом используется константа **dbLangCyrillic**.

Мы коснулись здесь только самых существенных моментов работы с БД через объекты DAO. Частности, как всегда, мы можем выяснять по справке VB5 в процессе решения практических задач.

Глава 4

КОМПОНЕНТЫ ACTIVEX

"В мире компонентов нет эквивалентов", как говорили старые алхимики, а они-то знали что говорили.

Покупайте советские часы — самые быстрые в мире.

Если читатель этой книги сумел вслед за автором добраться до этой главы и если в свое время он уже поучился программированию на обычном QBasic, PASCAL или C, то он уже "почувствовал разницу", о которой мы предупреждали во введении, между *обычным* и *Visual*-программированием. Говоря метафорически, "центр тяжести" приложения смещен в Visual-программировании в *объекты*; программный код, который пишет программист, является просто некоторым вспомогательным связующим звеном между объектами — компонентами приложения. При этом в Visual-языках естественным образом возникают целые библиотеки классов, участвующих в генерации объектов; *парадигма* программирования меняется настолько, что выражение "изучить язык Visual Basic (или Visual C++)" имеет смысл, довольно отличный от смысла выражения "выучить язык QBasic (или C)". Если в случае обычного алгоритмического языка его синтаксис и семантику (смысл синтаксических конструкций) можно было довольно просто описать с помощью естественного языка ("метаязыка") и этого было бы уже достаточно, чтобы, проявляя то или иное "искусство программирования" (в том числе и технологию "структурного программирования"), создавать приложения любой сложности (если отвлечься от вопроса о трудозатратах), то в Visual-программировании, помимо синтаксиса и семантики некоторого языка, необходимо овладение технологиями построения приложения из *компонентов*, соответствующим инструментарием, предоставляемым для этого интегрированной средой, и уже существующими наработками объектов.

Итак, в концепцию Visual-программирования заложено построение приложения из компонентов (хотя сама эта концепция возникла безотносительно к Visual-программированию). В основе такого подхода лежит некий стандарт, называемый COM *(Component Object Model)* — "*компонентно-объектная модель*". Эта модель реализована в VB5 в форме *технологии компонентов ActiveX*.

Компонент — это файл с расширением EXE, DLL или OCX, который содержит некоторый код со свойствами:

— *многократного использования* для обеспечения определенного набора функций;

— *динамического* (т.е. в фазе выполнения) подключения к приложению, называемому при этом *клиентом*, и отключения от него;

— обеспечения *интерфейса для связи с клиентом* — не зависимого ни от собственной реализации (платформы, языка), ни от реализации клиента.

В VB5 возможно создание *трех типов* компонентов ActiveX: *пользовательских элементов управления* ActiveX (тип проекта — **ActiveX Control**), *компонентов программ* ActiveX (типы проектов — **ActiveX DLL, ActiveX EXE**) и *документов* ActiveX (**ActiveX Document DLL, ActiveX Document EXE**). Рассмотрим данные типы компонентов подробнее.

Пользовательский элемент управления ActiveX (OCX)

Пользовательский элемент управления (ПЭУ) — это объект многократного использования. Так же как и компонент программ (*Code Component*), он *предоставляет приложению* свои свойства, методы и события, с помощью которых обеспечиваются его функциональные возможности, однако, в отличие от него, ПЭУ *размещается* (средствами визуальной компоновки) в приложении-клиенте (или другом ПЭУ), в то время как компонент программ, являясь приложением-сервером, просто предоставляет часть своих функций приложению-клиенту. Во всех случаях приложение-клиент должно, разумеется, поддерживать технологию ActiveX, являясь, например, продуктом, созданным фирмой Microsoft для работы под управлением Windows 9x (Office 97, Access 97, Internet Explorer, Visual FoxPro), либо просто любым приложением, созданным с помощью интегрированной среды VB5; можно использовать ПЭУ и в web-страницах.

Обычно ПЭУ включает в себя визуальную составляющую (которая, вообще говоря, необязательна) и программный код. Программный код и свойства ПЭУ хранятся в файле с расширением CTL (аналог файла формы FRM приложения VB5), который в фазе проектирования можно включать в любое приложение VB5. То же относится и к файлам графики (если ее элементы включены в ПЭУ), имеющим расширение CTX.

Таблица 4.1

Microsoft Windows Common Controls 5.0	TabStrip	— Вкладка (для многостраничного режима некоторой области окна приложения)
	Toolbar	— Панель инструментов (для рисунков на ее кнопках используется элемент **ImageList**)
	StatusBar	— Панель состояния (контейнер для элементов, отображающих текущее состояние приложения)
	TreeView	— Отображение элементов в виде древовидной структуры (как в Проводнике Windows)
	ListView	— Представление списка четырьмя способами — как в Проводнике (в правой части окна) Windows
	ImageList	— Контейнер для коллекции элементов **ListImage** (нумерация с 1), определяющих изображения
	Slider	— "Ползунок", подобный полосе прокрутки
	ProgressBar	— "Индикатор процесса", позволяющий контролировать зависание приложения и процент его завершенности
Microsoft Common Dialog Control 5.0	Common Dialog	Обеспечивает интерфейс между VB и динамической библиотекой **Commdlg.dll**. С помощью методов ShowColor, ShowFont, ShowHelp, ShowPrinter, ShowOpen, ShowSave открывает соответствующие стандартные диалоги (ShowHelp вызывает Windows Help Engine)
Microsoft Rich Textbox Control 5.0	RichTextBox	Поле ввода с расширенными возможностями форматирования, внедрения OLE-объекта, связью с полем типа **memo** БД
Microsoft Tabbed Dialog Control 5.0	SSTab	Набор вкладок (с ярлычками), каждая из которых может быть контейнером для других элементов управления
Microsoft FlexGrid Control 5.0	MSFlexGrid	"Сетка" для представления и редактирования табличных данных (строк и картинок). С элементом **Data** в качестве источника данных может функционировать только их просмотр
Microsoft PictureClip Control 5.0	PictureClip	Содержит растровые картинки, используемые для мультипликации, в форме одиночной картинки, из которой последовательно "вынимаются" сменяющие друг друга фрагменты
Microsoft Windows Common Controls-2 5.0	Animation	— Используется для анимации кнопок (по щелчку мышью) посредством воспроизведения AVI-файлов (без звука и сжатия или сжатые RLE-методом)
	UpDown	— "Счетчик", позволяющий перебирать числовые значения, отображаемые в другом элементе. Аналог Spin Button в VB4
Microsoft Internet Transfer Control 5.0	Inet	Производит поиск по заданному URL-адресу и отображает структуру каталогов и их содержание. Реализует протокол FTP для передачи файлов в Internet
Microsoft Internet Controls	WebBrowser	Включается в состав компонентов при установке Internet Explorer 3.0 и старше. Позволяет осуществлять поиск в Internet и отображение HTML-страниц в виде обычных web-страниц

(аналоги файлов FRX). Но обычно готовый ПЭУ компилируется в файл OCX, причем в одном файле может содержаться несколько ПЭУ. Подготовка ПЭУ к распространению (так же как и любого приложения) осуществляется с помощью программы Application Setup Wisard, входящей в состав VB5.

Мы уже познакомились с некоторыми ПЭУ, выбирая их из списка во вкладке **Controls** диалога **Components**, выводимого по соответствующей команде пункта главного меню **Project** либо по Ctrl T. Так, в первом варианте "Словарной обучалки" мы познакомились с элементом мультимедиа **MMControl**, во втором варианте — с элементами специализированных для БД списков **DBList** и **DBCombo** и сетки **DBGrid**. В таблице 4.1 приведены краткие описания еще семнадцати наиболее популярных ПЭУ из числа входящих в старшие редакции VB5 (Professional и Enterprise). В первом столбце таблицы приводится название компонента по списку во вкладке **Controls** диалога **Components**, во втором — имя класса элемента, под которым данный элемент появится в панели инструментов приложения, а также может быть найден в справочной системе для более подробного ознакомления.

В данной главе чуть позднее мы и сами разработаем свой ПЭУ, а затем используем его в демонстрационном проекте.

Компоненты программ ActiveX и технология OLE

ПЭУ ActiveX представляют собой один из четырех типов (будем считать, что *четвертый*) реализации *технологии OLE*, разработанной фирмой Microsoft в соответствии с идеологией COM-архитектуры. Рассмотрим другие три типа.

Технология OLE обеспечивала порядок использования в документе, созданном в одном приложении (например, в текстовом файле MS Word), объекта, созданного в другом приложении (например, картинки, созданной в графическом редакторе Paint). Первыми двумя типами технологии OLE были *связывание (Linking)* и *внедрение (Embedding)*.

При связывании объект из документа приложения-сервера ("источника") подключался к документу приложения-клиента ("приемника") с помощью ссылки (*Link* — "связь") в приемнике на объект в источнике. Никаких копий связанного объекта при этом не создается и все изменения объекта в документе клиента сохраняются и в документе сервера.

Связывание, таким образом, очень экономично по расходу внешней памяти, однако *теряется свобода перемещения* документа приложения-клиента на другой компьютер, так как объект из документа сервера не будет перемещен вместе с ним.

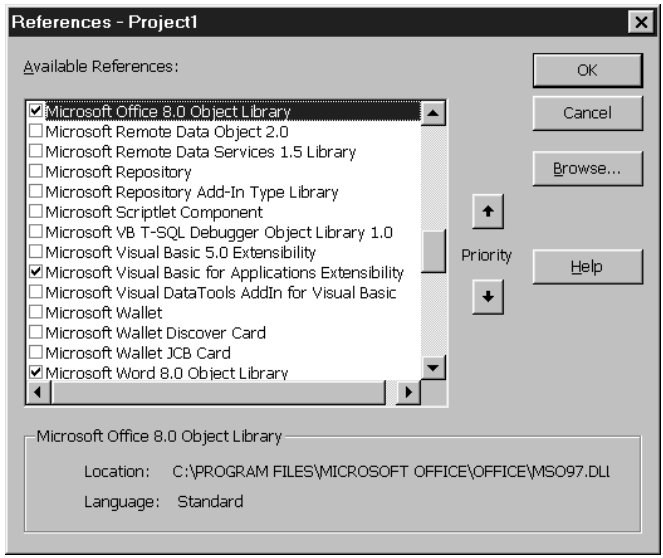
При внедрении создается копия объекта из документа сервера, которая и вставляется ("внедряется") в документ клиента. При этом расход ресурсов возрастает, зато появляется свобода перемещения документа клиента.

С внедрением по OLE-технологии мы уже познакомились при заполнении БД "Словарной обучалки" визуальными и звуковыми отображениями предметов, причем соответствующие поля имели, как мы помним, тип OLE. Кроме того, там же мы использовали специальный элемент управления OLE ("контейнер OLE", как еще его называют) для добавления в проект звукового сигнала. При этом в диалоге **Вставка объекта** в элемент OLE в случае выбора опции **Создание из файла** в диалоге появлялся переключатель **Связь**, включением которого можно было выбрать вставку объекта *связыванием* вместо внедрения. Интегрированный таким образом объект должен допускать "*активизацию по месту*": двойным щелчком мышью мы можем загрузить его в вызываемое приложение-сервер для возможности редактирования объекта.

Связывание и внедрение были первыми двумя типами OLE-технологии, реализующей принципы COM. Третьим типом явилась *автоматизация OLE* (OLE Automatisation), с которой, собственно, и начинается *технология компонентов ActiveX*.

Автоматизация OLE заключается в использовании приложением-клиентом объектов, их свойств и методов, предоставляемых приложением-сервером (то же: "*сервером автоматизации OLE*", "*компонентом программ ActiveX*"). Эти предоставляемые объекты описываются в *библиотеках OLE-автоматизации*, на которые для использования их объектов нужно устанавливать *ссылки*.

Необходимым (но не достаточным) условием такого использования объектов и методов приложения-сервера является его *регистрация в операционной системе* (осуществляемая обычно при инсталляции приложения). В интегрированной среде VB5 доступные библиотеки OLE-автоматизации могут просматриваться в диалоге **References** (Ссылки), вызываемом соответствующей командой из пункта **Project** главного меню:



После установления ссылки мы можем ознакомиться с *предоставляемыми* нужной библиотекой объектами, методами и свойствами (вызов диалога **Object Browser** по F2).

Существуют два типа взаимодействия клиента и сервера (и, соответственно, два типа серверов автоматизации OLE): *in-process (внутрипроцессный)* и *out-of-process* (*внепроцессный*). Внепроцессный сервер — это приложение, которое представляет *самостоятельно исполняемый файл* (находящийся в собственном пространстве адресов, вне процесса исполнения приложения-клиента), который активизируется клиентом, исполняемым в тот же момент в другом адресном пространстве. Если в VB5 создается такой внепроцессный OLE-сервер (компонент программ ActiveX), то выбирается проект ActiveX EXE, создающий исполняемый EXE-файл. Если же создается сервер, который будет исполняться в рамках процесса исполнения клиента, то выбирается проект ActiveX DLL (создание *библиотеки динамической компоновки*).

Разновидностью внепроцессного сервера является *удаленный* сервер, расположенный на другом компьютере. Такая удаленная OLE-автоматизация использует развитие COM-архитектуры, называемое DCOM — *Distributed COM (распределенная COM)*. Например, можно создать компоненты-серверы, обеспечивающие доступ к удаленной БД. Такие серверы (*серверы прикладных объектов — Business Object Server*)

широко используются в архитектуре *технологии "клиент—сервер"* для обеспечения *уровня бизнес-интерфейса* между клиентом (приложением-клиентом пользователя, обращающимся к БД) и сервером (БД системы клиент—сервер: Microsoft SQL Server, Oracle, Sybase).

Раз уж зашла об этом речь, попытаемся описать технологию клиент—сервер в двух словах.

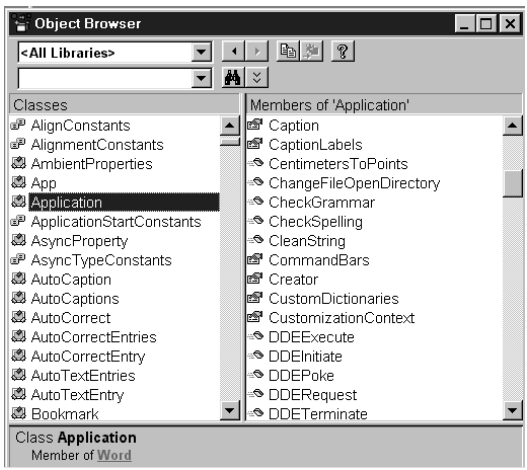
Базы данных создаются чаще всего не для автономного пользователя, а для целого коллектива, ну, например, фирмы. Это обуславливает объединение всех компьютеров в сеть, причем один (или не один) из компьютеров в этой сети является файлом-сервером, содержащим БД. Управляет доступом пользователей к БД *сетевая операционная система*, а описанная модель совместной обработки данных называется *сетевой информационной системой* (ИС). Именно ей на смену и пришла модель многопользовательской системы "клиент—сервер".

Дело в том, что сетевая ИС страдает по меньшей мере двумя серьезными недостатками. Во-первых, каждый пользователь, обращаясь к таблице БД, делает ее на это время недоступной для других пользователей. Во-вторых, данные, с которыми работает пользователь, копируются при этом на его компьютер, что предопределяет высокую загруженность такой сети.

В системах, работающих по технологии "клиент—сервер", вместо запроса данных у сервера и последующей обработки на компьютере-клиенте клиент посылает серверу запрос (мы уже знакомились с этим понятием; добавим только, что запрос клиента — это в большинстве своем обращение к "*пользовательским хранимым процедурам*", которые, будучи созданы пользователем на языке SQL, хранятся на сервере БД в откомпилированном, готовом к выполнению виде) на получение определенного результата, а сервер, обрабатывая его, сам производит обработку данных БД и возвращает клиенту требуемый результат. Это — двухуровневая архитектура "клиент—сервер": первый уровень — клиент, второй — клиент-серверная БД. При *трехуровневой* архитектуре между этими двумя уровнями включается уровень бизнес-интерфейса, обеспечиваемый как раз сервером удаленной OLE-автоматизации (сервером прикладных объектов). Его задача, во-первых, свести в одну точку (поэтому он располагается на стороне сервера БД) и стандартизировать процедуру обращения к серверу БД с клиентской стороны и, во-вторых, освободить клиента от учета тех или иных особенностей типа БД, с которой он работает. Общение клиента с сервером БД производится, таким образом, через программирование на сервере удаленной автоматизации свойств объектов уже упоминавшейся библиотеки RDO *(Remote Data Objects)*. Утилиты VB5, обеспечивающие возможности DCOM, входят только в состав редакции **Enterprise** и здесь не рассматриваются.

Библиотеки OLE-автоматизации (иначе — *библиотеки типов*) могут быть ресурсами и внутри-, и внепроцессного OLE-серверов. Могут эти библиотеки и входить в файлы документов, и иметь форму *независимых двоичных файлов* (расширения OLB или TLB). Все эти варианты встречаются среди компонентов программ, представленных в приведенном списке диалога **References**. (К сожалению, библиотеки подобного типа имеются не у всех серверов OLE-автоматизации.)

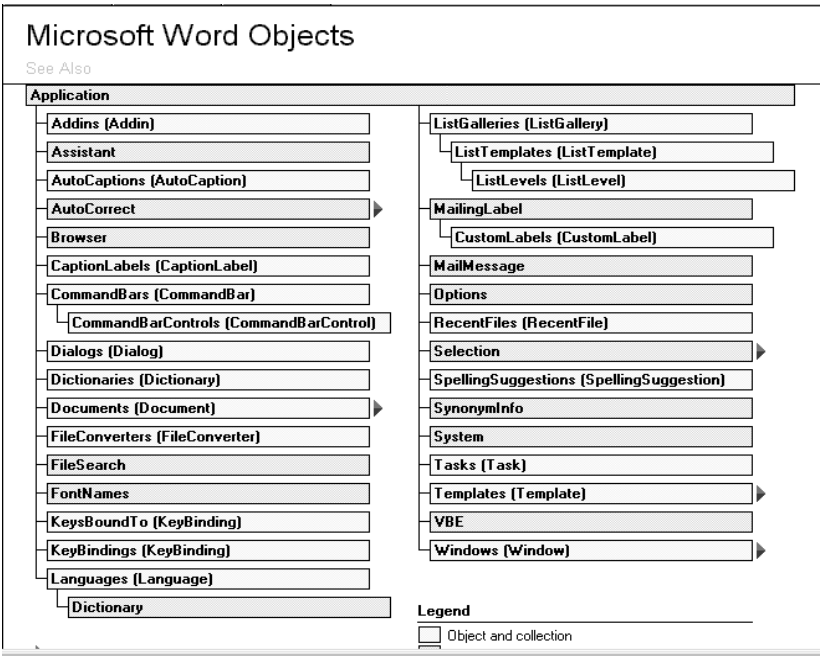
Объекты, предоставляемые сервером OLE-автоматизации, могут структурироваться в *иерархическую объектную модель*, отражающую, какие объекты сервера входят в состав других объектов (уточним еще раз: говоря в данном контексте об "объектах", мы имеем в виду, разумеется, *классы*, порождающие соответствующие объекты; разница между объектом и его классом та же, что между детским куличиком и формочкой для его изготовления). Рассмотрим для примера объектную модель приложения Microsoft Word 97. Вызовем **Object Browser** (F2) и посмотрим в список **Classes** (Классы) при выбранном пункте **<All Libraries>** в списке библиотек (слева сверху). В данном списке могут присутствовать (а могут и не присутствовать) несколько объектов (классов, конечно) с именем **Application**. Выбирая их мышью, мы можем видеть названия представляемых объектом **Application** приложений в нижней строке диалога. Так, про объект **Application**, представляющий "верхний уровень" приложения Word, будет сказано: **Member of Word (член <объекта> Word)**:



Если такого объекта **Application** в списке еще нет, то установим на него ссылку. Щелкнув в поле диалога правой клавишей мыши, выберем

в контекстном меню известную нам команду **References** (дублирующую одноименную команду пункта **Project** главного меню). В появившемся диалоге с именами библиотек OLE-автоматизации включим переключатель в строке **Microsoft Word 8.0 Object Library** и выполним команды диалога **Применить** и **OK**. Нужный объект появится в списке окна **Object Browse**r.

Если при этом открыть верхний слева список библиотек, то можно увидеть в нем новый пункт — **Word**. Выбрав его, мы получим в списке **Classes** список классов именно этой библиотеки, причем для каждого выбранного класса в правом списке будут отображаться его члены (свойства, методы и события). После выбора представляющего Word объекта **Application** в диалоге **Object Browser** вызовем справку (F1), в которой будет графически представлена объектная модель данного сервера OLE-автоматизации:



Понятно, что справочная система обеспечивает нам возможность пройти по любой ветви данного дерева до любого ее “листика”.

Сделаем ряд важных замечаний относительно обращения с объектной моделью сервера.

- В некоторых узлах дерева представлены не простые объекты, а объекты-коллекции *(Collection Object)*.

  Коллекция — это объект, *объединяющий другие объекты* ("входящие в него"). Коллекция имеет свои свойства и методы, не зависимые от свойств и методов входящих в нее объектов. Объекты, входящие в коллекцию, могут иметь разный тип. Обращаются к ним, используя аппарат индексов (возможны и другие способы — по имени, например).

  В VB5 есть две встроенные коллекции: **Forms**, представляющая все формы проекта, и **Controls** — все его элементы управления.

- Для обращения к объектам нижележащего уровня, а также к их членам (свойствам и методам), используют навигационную точку точно таким же образом, как при обращении к элементам управления, содержащимся в других элементах управления, и их членам. При этом в качестве головного имени используется имя библиотеки (в нашем случае — Word согласно списку библиотек в **Object Browser**) либо квалификатор **Application**, если этот объект единственный среди всех библиотек (согласно списку **Classes** в **Object Browser**), на которые были установлены ссылки в данном проекте. Использовать же для обращения к объектам, зависимым от объекта **Application**, навигационную цепочку вида: **Word.Application** нет необходимости: квалификатор **Application** можно не использовать. Дело в том, что все свойства объекта **Application**, возвращающие зависимые объекты, являются одновременно глобальными членами библиотеки (пункт <**globals**> в списке **Classes**). Поэтому, например, для обращения к методу **Add** коллекции **Documents**, открывающему новый документ приложения Word, достаточно следующего предложения:

```
Word.Documents.Add
```

  — или, если коллекция **Documents** единственна в списке глобальных членов при выбранном пункте <**All Libraries**> списка библиотек, то просто

```
Documents.Add
```

Итак, мы установили ссылку на сервер и научились получать информацию о его объектной модели и ее элементах. Займемся программированием использующего сервер приложения-клиента.

Работа с серверными объектами в приложении-клиенте

После установки ссылки на сервер OLE-автоматизации (он же компонент программ ActiveX, он же сервер прикладных объектов) в приложении-клиенте нужно выполнить еще два шага, чтобы получить наконец возможность работать с серверными объектами. Первый из этих шагов — *объявление объектной переменной*, которая будет ссылаться на объект-сервер, второй — *создание в клиенте объекта-сервера* (в смысле экземпляра, а не класса!). Через свойства и методы головного объекта-сервера мы получаем далее доступ и к зависимым его объектам. При этом *для обращения* к многократно используемым зависимым объектам можно объявлять объектные переменные и устанавливать с их помощью *ссылки на эти объекты*.

Мы опишем шаги по созданию в клиенте доступа к OLE-серверу на примере сервера Word 97, помня, однако, что все описанное, например, про объявление объектной переменной, относится в той же мере вообще к любым, то есть и "внутренним", объектам любого приложения.

❐ Объявление объектных переменных имеет две основные формы. В первой из этих форм объявляется "общая" объектная переменная типа **Object** (как и во всех случаях, можно использовать и тип **Variant**), которая может ссылаться на объект любого типа (класса), например:

```
Dim objVar As Object
```

Во второй форме объявляется объектная переменная *определенного* объектного класса, которая может ссылаться только на объекты того же типа (класса), например:

```
Dim objVar As Word.Selection
```

Такой вариант объявления является предпочтительным, так как, во-первых, компилятор обнаружит связь переменной с классом еще во время написания кода приложения-клиента и обеспечит программисту сервис **List Members** (список членов объекта в ответ на ввод навигационной точки после имени переменной), а во-вторых, компилятор использует в этом случае *раннее связывание* клиента с объектом (во время компиляции, а не выполнения, как при *позднем*), что предпочтительнее для отладки и по быстродействию.

- Создание экземпляра объекта—компонента программ (OLE-сервера) производится в клиенте *одновременно с установкой на него ссылки* для предварительно объявленной переменной объектного типа (общей либо того же класса). Установление ссылки переменной на объект-сервер (существующий или создаваемый) производится инструкцией **Set**, поэтому все 3 варианта создания экземпляра сервера являются вариантами этой инструкции.

  Почему объект не может быть создан без установления на него ссылки? Дело в том, что по COM-спецификации объект должен *вести счет* устанавливаемым на него ссылкам, с тем чтобы при нулевом их количестве (когда в объекте генерируется событие **Terminate** — "уничтожение") уничтожить себя. Поэтому созданный объект должен иметь по крайней мере *одну* ссылку.

  Принудительное зануление количества ссылок путем присвоения ссылочной переменной значения **Nothing** приводит к уничтожению объекта.

  Первый вариант использует для создания объекта (**Selection**, например) функцию **CreateObject**:

```
Set objVar = CreateObject("Word.Selection")
```

  Второй вариант использует ключевое слово **New**:

```
Set objVar = New Word.Selection
```

  Таким же образом можно создавать и несерверные объекты.

  Третий вариант использует вызов функции **GetObject** с пустой строкой в качестве значения первого аргумента, который должен специфицировать файл, содержащий объект определенного вторым аргументом класса (либо, при отсутствии второго аргумента, объект, ассоциированный с расширением данного файла):

```
Set objVar = GetObject("", "Word.Selection")
```

  (Если бы первый аргумент был опущен, данная инструкция устанавливала бы ссылку на уже существующий объект выполняемого приложения Word либо, при отсутствии последнего, генерировала бы ошибку.)

  Отметим, что если тип сервера OLE-автоматизации внепроцессный, как в случае приложения Word, то для создания в клиенте его объектов он должен быть открыт (запущен на выполнение). Сделать это можно путем создания его объекта **Application**.

  Существует еще один способ создания объекта с использованием ключевого слова **New** в объявлении объектной переменной, однако ввиду его неэффективности он не рассматривается.

В качестве примера, иллюстрирующего процесс создания доступа в приложении-клиенте к серверу OLE-автоматизации, создадим стандартный проект, который по "нажатию" командной кнопки на форме будет загружать Word 97 и открывать в нем существующий документ, заданный своим полным именем (с путем). Весь код такого приложения будет содержаться в обработчике события **Click** для указанной кнопки:

```
Private Sub cmdWord_Click()
 Dim objWord As Object 'Объявляем объект
 On Error Resume Next 'Обратите внимание на эту инструкцию:
 '"В случае ошибки (On Error) продолжать (Resume) со
 ' следующей (Next) инструкции"
 'Открываем объект, установив ссылку на активный объект Word:
 Set objWord = GetObject(, "Word.Application")
 'Если такого объекта не оказалось,
 If objWord Is Nothing Then
  'то открываем его, запустив Word на выполнение:
  Set objWord = New Word.Application
  '(То же можно сделать, например, так:
  'Set objWord = CreateObject("Word.Application")  )
  'Если и после этого ссылка не удалась,
  If objWord Is Nothing Then
   'то распишемся в своем бессилии:
   MsgBox "Создать объект не удается!"
  End If
 End If
 objWord.Visible = True ' Иначе окна приложения не увидим!
 'Откроем, пожалуй, наш документ:
 objWord.Documents.Open "C:\Мои документы\MyDoc.doc"
 'Уничтожаем объект ("Знак хорошего вкуса и традиций пример"):
 Set objWord = Nothing
End Sub
```

Для завершения работы с Word 97 в клиенте необходимо использовать метод **Quit** объекта **Application**.

Конечно, мы рассмотрели и проиллюстрировали здесь только *принципиальную* сторону OLE-автоматизации на компоненте Word 97.

VB5 позволяет нам не только использовать в приложениях-клиентах чужие повторно используемые объекты (классы), но и создавать приложения-серверы. В любом из 7 типов проектов VB5, кроме стандартного EXE, можно создавать такие повторно используемые классы, объекты которых создает и использует клиент. Повторную используемость класса (объявленного обязательно как **Public**) определяет наличие у него свойства **Instancing** (используемость).

Сейчас мы научимся *создавать* классы, а потом и превращать их при надобности в *повторно используемые* в приложениях-клиентах.

Учимся создавать класс

Классы — это функционально обособленные конструкции, которые описывают абстрактные "объекты". Идея классов легла в основу объектно-ориентированного программирования (ООП), пришедшего на смену процедурному. Почему это произошло?

Раньше каждая процедура, назначением которой было выполнение определенной функции в рамках алгоритмической реализации решения задачи, работала с некоторым подмножеством множества всех данных программы. Эти подмножества пересекались между собой произвольным образом. Когда задачи и программы усложнились (вслед за стремительным ростом отношения производительность/стоимость микропроцессоров), начался резкий рост проблем поддержки программного продукта: изменения в составе *данных* программы вели к изменению заранее непредсказуемого количества зависимых от этого состава *процедур*. В ООП данные вместе с процедурами хранятся в *классе* и, как правило, *недоступны извне*. Это качество класса называется *инкапсуляцией*. Когда объекты, построенные на основе классов, обмениваются между собой сообщениями (в VB5 — через генерацию событий, изменение значений свойств и вызов методов, как мы знаем), то, реагируя на них, они могут изменять *только свои собственные* данные (и посылать, в свою очередь, сообщения другим объектам).

Идея класса оказалась прекрасно вписываемой в COM-архитектуру. Дело в том, что компонентам COM также присуща инкапсуляция: клиент должен подключаться к компоненту через *интерфейс*, который не должен зависеть ни от реализации клиента, ни от реализации сервера.

Мы создадим сейчас класс в форме *модуля класса* — одного из типов программных модулей, который добавляется к стандартному проекту (до сих пор мы работали только с *модулями экранных форм*).

Итак, открыв стандартный проект, выполним команду **Add Class Module**, входящую в пункт **Project** главного меню. В появившемся окне на вкладке **New** запустим функцию создания пустого класса **Class Module**. Мы оказываемся в окне кода модуля. Как обычно, вверху окна слева располагается список **Object**, содержащий название секции объявлений **(General)** и названия объектов (у нас пока единственный "объект" с названием **Class**), а справа — список **Procedure**, содержащий названия процедур, относящихся к тому или иному "объекту". Так как в окне кода содержится код, относящийся всегда к одному модулю класса, то вместо реального имени класса в списке **Object** указывается просто **Class** (так же, как в окне кода формы вместо реального имени формы — просто **Form**), что вряд ли можно считать удачной находкой разработчиков

среды VB5. Для "объекта" **Class** в правом списке уже присутствуют названия (а в окне кода вызываются заготовки соответствующих обработчиков) двух обязательных событий: **Initilize** (инициализация) и **Terminate**, которые *возбуждаются* объектом данного класса при его создании и уничтожении соответственно (в основном эти события используются в обработчиках тех программных модулей, где создаются объекты класса).

Класс, который мы хотим сейчас создать, должен быть шаблоном для объекта, реализующего *стек*. Этот пример хотя и не будет простым, зато хорошо проиллюстрирует технику работы с объектами, в том числе и внутри порождающего их класса ("рекурсивные объекты").

Определим для начала понятие "стек" (а заодно и "очередь").

Пусть мы имеем множество каких-либо элементов, которое образовалось следующим образом. Сначала во множестве был один элемент. Потом к этому множеству добавили второй элемент следующим способом: в первый элемент поместили ссылку на второй (ссылка — это связь через адрес). И потом все время добавляли во множество $i+1$-й элемент помещением ссылки на него в i-й элемент (ссылка на последующий элемент). Множество, образуемое таким способом, называется однонаправленным списком, точнее, его разновидностью, называемой *очередью*. Ну а если добавлять $i+1$-й элемент путем помещения в него ссылки на i-й (на предшествующий), то получится другая разновидность однонаправленного списка — *стек* (*stack* — "кипа"). (Ну а если добавлять сразу обе указанные ссылки, то получим двунаправленный список.)

Множество, являющееся списком, весьма сильно отличается от обычного множества с индексированными элементами. В множестве с индексом мы в любой момент можем обратиться по индексу к любому его элементу. В списке же мы можем обращаться к элементу только оттуда, откуда есть ссылка на этот элемент.

В очереди нам доступна ссылка из первого элемента на второй, а первый элемент доступен непосредственно, так как он "крайний", "в вершине" списка. В стеке, наоборот, в вершине всегда находится последний элемент, из которого доступен предпоследний, и т.д.

Из сказанного следует, что стек заполняется с *вершины*, причем каждый новый элемент, становясь в вершину, "проталкивает" бывший в вершине элемент, а за ним и все остальные, "в глубь" стека. Разумеется, это проталкивание происходит на логическом уровне, физического перемещения данных при этом не происходит, все элементы остаются на прежних местах.

В качестве элементов стека естественно использовать объекты, которые, с одной стороны, могут включать в себя в форме значений свойств любые данные, а с другой стороны — имеют доступ к себе посредством ссылок (через ссылочные переменные и свои свойства).

Создаваемый класс (назовем его **Stack**) будет представлять объект (назовем его *текущим*), который всегда является вершиной стека. У этого объекта имеются свойства **Value** (хранимое в элементе стека данное) и **Child** (возвращающее объект того же типа (класса), а также два метода: **Push** ("протолкнуть") и **Pop** ("выскочить"). Первый из них увеличивает глубину стека, создавая новый элемент в его вершине. Делается это так: 1) создается новый объект типа **Stack**; 2) его свойству **Child** устанавливается ссылка на объект, возвращаемый свойством **Child** текущего объекта, а в свойство **Value** копируется значение **Value** текущего объекта; 3) свойству **Child** текущего объекта устанавливается ссылка на новый объект. Таким образом, новый объект забирает у текущего хранимое в нем данное и как бы вставляется между текущим объектом и тем, на который тот ссылался до того свойством **Child**. Таким образом и происходит как бы "проталкивание" стека.

Метод **Pop** реализует обратный к методу **Push** сдвиг стека в направлении его вершины. Происходит это путем уничтожения объекта, на который ссылается текущий в свойстве **Child** (после копирования в текущий данных из этого сыновнего объекта). Перед этим ссылка на этот сыновний объект устанавливается для вспомогательной объектной переменной **objElem**, через которую текущему объекту передается ссылка в свойстве **Child** на "внучатый" объект, следующий за бывшим сыновним.

В описанном здесь алгоритме вроде бы не предусмотрено создания никаких событий, возбуждение которых полезно было бы обрабатывать в содержащем объект приложении. Однако принято, чтобы по крайней мере события **Initilize** и **Terminate**, действующие внутри класса, могли возбуждаться классом и в приложении. Кроме того, мы создадим событие **Void**, возбуждаемое тогда, когда стек оказывается пуст, т.е. делается попытка "выталкивания" единственного оставшегося от него объекта.

Таково устройство и принцип работы проектируемого нами класса. Теперь изложим последовательность работы по созданию его свойств, событий и методов.

- Первое, что мы сделаем, — это объявим в создаваемом классе **Stack** объектную переменную **objElem** типа **Stack**, которая позволит *объекту* создавать *объекты того же класса*, порождая, таким образом, элементы стека:

```
Option Explicit 'Контроль за явной объявленностью
                'переменных
Private objElem As Stack
```

- Создать свойство можно двумя способами:

  — описать переменную с именем и типом свойства класса **Public**;

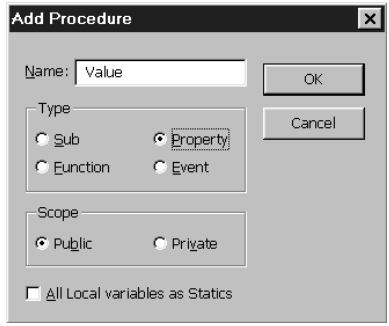
  — использовать процедуры свойств **Property**.

  При использовании первого способа переменная-свойство доступна после создания объекта данного класса в приложении и для чтения, и для записи, причем контроль за корректностью устанавливаемых значений невозможен. Второй способ более гибок. Во-первых, процедуры свойств позволяют разделить функции чтения и записи значений. Так, процедура **Property Get** используется для считывания значений свойства, а процедура **Property Let** — для установки значения из приложения (если свойство представлено объектной переменной, то вместо процедуры **Property Let** используется **Property Set**). Во-вторых, использование процедур свойств позволяет кодировать проверку корректности устанавливаемых свойству значений.

  Свойство **Child** мы зададим *первым* из описанных способов, описав в разделе деклараций **(General)**:

```
Public Child As Stack
```

  Для задания свойства **Value** вызовем командой **Add Procedure** пункта **Tools** главного меню соответствующий диалог и установим в нем имя свойства **Value**, а также опции **Property** и **Public**:



После нажатия на **OK** мы увидим в окне кода две заготовки:

```
Public Property Get Value() As Variant
End Property
Public Property Let Value(ByVal vNewValue As Variant)
End Property
```

Конечно, сами эти заготовки не способны хранить значение свойства. Для этого в декларациях объявляют специальную переменную уровня модуля:

```
Private mvar_Value As Variant
```

После этого в процедуру **Property Get Value** помещается следующая инструкция, обеспечивающая *считывание* из приложения значения хранимого свойства:

```
Value = mvar_Value
```

(Заметим, что если бы мы создавали свойство только для чтения, то заготовку **Property Let Value** надо было бы удалить, а вместо специальной переменной использовать для присвоения значения свойству константу, представляющую это значение.)

В процедуру **Property Let** свойство **Value** передает "по значению" (ключевое слово **ByVal**) через параметр **vNewValue** *новое значение свойства*. Его мы, естественно, запоминаем в нашей специальной переменной. Таким образом, процедуры свойств (вместе с добавкой объявления **mvar_Value**) приобретают следующий "стандартный" вид:

```
Public Property Get Value() As Variant
 Value = mvar_Value
End Property

Public Property Let Value(ByVal vNewValue As Variant)
 mvar_Value = vNewValue
End Property
```

- Говоря о *создании событий класса*, важно подчеркнуть различие между событиями *возникающими* (внутри класса) и событиями *возбуждаемыми* (в приложении, где создан объект класса). Когда говорят о создании событий класса, имеют в виду именно *возбуждаемые* события.

  Для создания события класса нужно:
  - объявить событие (**Public Event** ***Имя_события***);
  - возбудить событие (**RaiseEvent** ***Имя_события***) в том месте кода модуля, где это требуется.

  В объявлении события после его имени могут располагаться круглые скобки с объявлениями параметров события. Тогда при возбуждении события стоящие на местах параметров аргументы будут переданы в приложение.

В разделе деклараций поместим объявления создаваемых нами событий:

```
Public Event Initilize()
Public Event Terminate()
Public Event Void()
```

После этого в заготовки обработчиков событий **Initilize** и **Terminate** (событий, получаемых модулем класса от операционной системы) помещаем инструкции **RaiseEvent Initilize** и **RaiseEvent Terminate** соответственно. Что же касается инструкции **RaiseEvent Void**, то ее мы поместим в тело инструкции **If**, проверяющей внутри процедуры **Pop** наличие у *текущего* объекта ссылки на его *сыновний* (если таковой нет, то событие вызывается, а процедура **Pop** заканчивает работу по инструкции **Exit** — "выход").

- Методы **Push** и **Pop** представляют собой обычные процедуры уровня доступа **Public**, реализующие описанный алгоритм. Приведем их в составе всего кода запрограммированного нами класса **Stack**:

```
Option Explicit
Private objElem As Stack
Public Child As Stack
Private mvar_Value As Variant
Public Event Initilize()
Public Event Terminate()
Public Event Void()
Public Property Get Value() As Variant
 Value = mvar_Value
End Property
Public Property Let Value(ByVal vNewValue As Variant)
 mvar_Value = vNewValue
End Property
Public Sub Push()
 Set objElem = New Stack 'Создаем новый объект со ссылкой objElem
 Set objElem.Child = Me.Child 'Ссылаемся сыном нового объекта
                              'на сына текущего объекта
 objElem.Value = Me.Value 'Копируем данное из текущего-в новый
 Set Me.Child = objElem 'Ссылаемся сыном текущего на новый
End Sub
Public Sub Pop()
 If Me.Child Is Nothing Then
  RaiseEvent Void
  Exit Sub
 End If
 Set objElem = Me.Child 'Сохраняем ссылку на сына
 Set Me.Child = Nothing 'Уничтожаем сына текущего объекта
 Set Me.Child = objElem.Child 'На место сына подставляем внука
 Me.Value = objElem.Value 'В текущий копируем данное, бывшее
Set objElem = Nothing 'в уничтоженном сыне, и уничтожаем
End Sub               'ссылку на этого сына
Private Sub Class_Initialize()
 RaiseEvent Initilize
End Sub
Private Sub Class_Terminate()
 RaiseEvent Terminate
End Sub
```

В коде методов обратите внимание на две новые конструкции.

Ключевое слово **Me** выполняет роль неявно описанной переменной, ссылающейся на текущий объект (уже встречалось в инструкции **Unload** кода формы, созданной для заполнения БД).

В инструкции **If** метода **Pop** в качестве условия используется операция ссылки на объект **Is**. Она возвращает **True** только тогда, когда левый и правый операнды ссылаются на один и тот же объект. В данном случае правым операндом выступает ключевое слово **Nothing**, означающее отсутствие объекта; следовательно, для истинности условия необходимо, чтобы и левый операнд не ссылался ни на какой объект.

Теперь протестируем созданный класс.

Для этого используем форму нашего стандартного проекта. Разместим на ней 3 кнопки: **Создать объект, В стек** и **Из стека**. Естественно, первая из них будет создавать объект класса **Stack**, а две другие — при каждом нажатии вызывать методы **Push** и **Pop** объекта. В обработчике события **Void** будет выдаваться соответствующее сообщение **(Стек пуст)**.

Правда, сделав все как надо, вы с удивлением обнаружите, что объект создается, методы его работают, но в списке объектов окна кода формы имя объекта **(Elem)** не появляется и, соответственно, в списке событий для этого объекта события **Void** нет. Дело в том, что объекты, *имеющие события*, должны объявляться с ключевым словом **WithEvents**. Вот код нашего теста:

```
Dim WithEvents Elem As Stack
Dim i As Integer, j As Integer
'i - натуральный ряд, заносимый в стек
'j - флаг, устанавливаемый при чтении из пустого стека
Private Sub cmdClass_Click()
 Set Elem = New Stack
 Elem.Value = -1 'При пустом стеке объект хранит -1
 cmdClass.Enabled = False
End Sub
Private Sub cmdPop_Click()
 Elem.Pop
 If j = 1 Then
  MsgBox Str(Elem.Value)
  i = i - 1
 End If
End Sub
Private Sub cmdPush_Click()
 If j = 0 Then
  j = 1
 End If
 i = i + 1
 Elem.Push 'При каждом вызове в стек заносятся числа
 Elem.Value = i
End Sub
Private Sub Elem_Void()
 MsgBox "Стек пуст"
 j = 0
End Sub
```

При первом "проталкивании" в стек заносится содержащаяся в объекте —1, при последующих — числа 1, 2, 3... и так далее. В любой момент эти числа можно извлекать из стека в обратном порядке, что контролируется выдачей их на печать.

Сохраним наш проект в файле **Pr_Class.vbp**, модулю класса присвоим имя (при сохранении) **Stack.cls**.

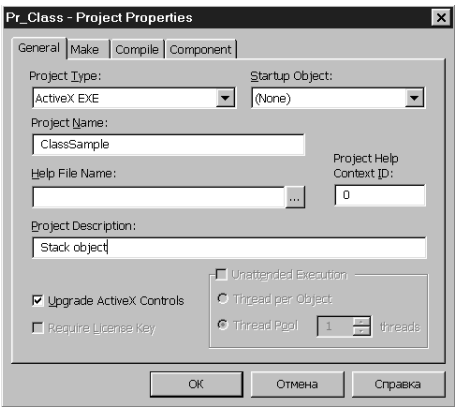
Помимо такого создания модуля класса "вручную", существует возможность обратиться к мастеру VB **Class Builder** (Построитель классов), запускаемому с помощью пиктограммы на уже известной нам вкладке **New** окна **Add Class Module**. С его помощью можно модифицировать и уже существующий класс.

Создаем компонент программ ActiveX EXE

Итак, мы имеем модуль класса **Stack.cls**, который был добавлен к проекту с именем **Pr_Class.vbp** и который описывает класс **Stack**. Указанный проект имеет *стандартный* тип, поэтому наш класс известен пока только в рамках данного проекта; в списке диалога **References** строки с его описанием нет.

Чтобы превратить проект со стандартным модулем класса в *компонент программ ActiveX EXE*, выполним следующие действия.

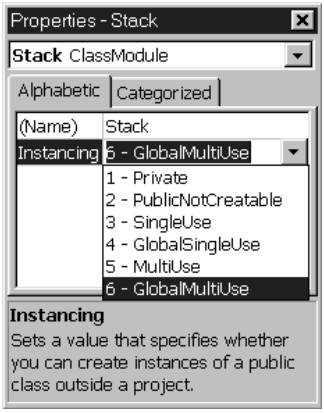
- Оставим в проекте только модуль класса, удалив файл формы (выделяем его в окне проекта, затем щелчок правой клавишей, затем команда **Remove** контекстного меню).
- Изменим тип проекта. Для этого откроем диалог **Project Properties**: **Project ⇨ Pr_Class Properties**. Во вкладке **General** диалога:
  - в списке **Project Type** выберем **ActiveX EXE**;
  - в списке **Startup Object — None** (отсутствует);
  - в поле **Project Name** введем новое имя проекта **ClassSample**;
  - в поле **Project Description** (Описание проекта) — **Stack Object**:



Заметим, что именно установленное здесь описание проекта — **Stack object** — будет появляться в списке библиотек типов диалога **References** с момента регистрации компонента в системе. Регистрация же эта произойдет *после компиляции*. Имя, данное проекту — **ClassSample**, — войдет при этом в список библиотек броузера объектов; в списке классов броузера будет имя класса **Stack**.

Перейдя далее на вкладку **Component** диалога **Project Properties**, в группе радиокнопок **Start Mode** включим опцию **ActiveX** (если она еще не была включена).

- После описанного изменения типа проекта в окне его свойств, помимо имевшегося у класса **Stack** свойства **Name**, появляется уже упоминавшееся свойство класса **Instancing** (используемость). На *рис.* 4.1 приведены все возможные (в проекте **ActiveX EXE**) значения этого свойства. Для нашего класса установим значение этого свойства равным 5 **(MultiUse)**.



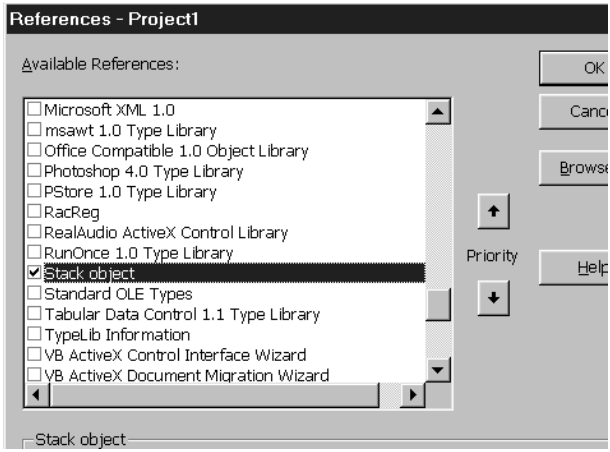
1 — клиент не может использовать класс;
2 — может использовать объекты, созданные самим сервером;
3 — каждый созданный клиентом объект запускает новый экземпляр сервера;
4 — как для 3, но свойства и методы класса могут использоваться клиентом как глобальные функции;
5 — один экземпляр сервера обеспечивает любое количество объектов для любого количества клиентов;
6 — как для 5, но сервер может создаваться автоматически при вызове метода или свойства класса.

Рис. 4.1. Значения свойства **Instancing**

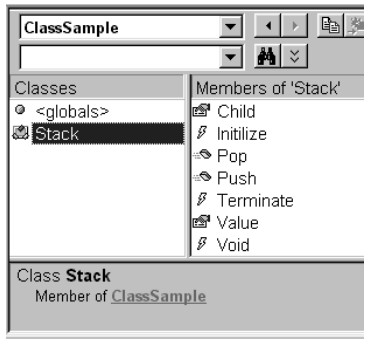
- Компилируем компонент: **File ⇨Make Pr ⇨Class.exe**. Создаваемому исполняемому файлу компонента даем перед компиляцией имя **ClassStack.exe**. При компиляции компонент регистрируется в системе и для него создается библиотека типов с именем **ClassSample**.

Перейдем теперь к тестированию созданного компонента.

Закроем проект с компонентом и откроем новый стандартный проект **Project1**. В нем вызовем диалог **References** и установим ссылку на описание созданного класса:



Вызвав после этого **Object Browser**, убедимся, что класс **Stack** доступен новому проекту вместе со всеми своими членами:



Для дальнейшего тестирования компонента нам достаточно добавить к новому проекту ту же форму, которая была создана в проекте **Pr_Class** для тестирования класса. Для этого мы сначала удалим (с помощью контекстного меню в окне проводника проекта) текущую форму **(Remove)**, а затем через цепочку **Add ⇨ Add Form ⇨ Existing** выйдем в диалог, позволяющий выбрать файл добавляемой экранной формы. После этого в свойствах проекта необходимо указать добавленную форму как **Startup Object** и запускать проект на выполнение. При нажатии на кнопку **Создать объект** загруженной формы произойдет запуск экземпляра сервера — компонента программ ActiveX, предоставляющего объект **Stack** и его члены.

Подобным образом можно создать и *внутрипроцессный* компонент программ (ActiveX DLL), для класса которого свойство **Instancing** по понятным причинам не может иметь значений 3 и 4.

ОБЪЕДИНЕНИЕ ПЕДАГОГИЧЕСКИХ ИЗДАНИЙ «ПЕРВОЕ СЕНТЯБРЯ»

Первое сентября (А.С. Соловейчик) индекс подписки — 32024; **Английский язык** (Е.В. Громушкина) индекс подписки — 32025; **Биология** (Н.Г. Иванова) индекс подписки — 32026; **Воскресная школа** (монах Киприан (Ященко)) индекс подписки — 32742; **География** (О.Н. Коротова) индекс подписки — 32027; **Здоровье детей** (А.У. Лекманов) индекс подписки — 32033; **Информатика** (С.Л. Островский) индекс подписки — 32291; **Искусство** (Н.Х. Исмаилова) индекс подписки — 32584; **История** (А.Ю. Головатенко) индекс подписки — 32028; **Литература** (Г.Г. Красухин) индекс подписки — 32029; **Математика** (И.Л. Соловейчик) индекс подписки — 32030; **Начальная школа** (М.В. Соловейчик) индекс подписки — 32031; **Немецкий язык** (М.Д. Бузоева) индекс подписки — 32292; **Русский язык** (Л.А. Гончар) индекс подписки — 32383; **Спорт в школе** (Н.В. Школьникова) индекс подписки — 32384; **Управление школой** (Н.А. Широкова) индекс подписки — 32652; **Физика** (Н.Д. Козлова) индекс подписки — 32032; **Химия** (О.Г. Блохина) индекс подписки — 32034; **Школьный психолог** (М.Н. Сартан) индекс подписки — 32898.

Гл. редактор
С.Л.Островский

Зам. гл. редактора
Е.Б.Докшицкая

Дизайн и компьютерная верстка:
Н.И.Пронская

Корректоры:
Е.Л. Володина,
С.М.Подберезина

Учредитель:
ООО “Чистые пруды”

Выходит четыре раза в месяц
При перепечатке ссылка на ИНФОРМАТИКУ обязательна, рукописи не возвращаются
Регистрационный номер 012868

Отпечатано в типографии ОАО ПО “Пресса-1”, 125865, ГСП, Москва, ул. Правды, 24
Тираж 5000 экз.
Заказ №

ИНДЕКС ПОДПИСКИ
для индивидуальных подписчиков — 32291
для организаций и предприятий — 32591
комплекта приложений — 32744

121165, Москва, Киевская, 24
Тел. 249 4896
Отдел рекламы
Тел. 249 9870

Тел. (095)249 3138, факс (095)249 3184

internet:inf@1september.ru
WWW:http://www.1september.ru

10.07