

А. Горев, С. Макашарипов, Р. Ахаян. **Эффективная работа с СУБД**

Часть 1. Что надо знать для создания системы обработки данных

Глава 1. Постановка задачи и разработка бизнес-правил

- 1.1. Некоторые определения
- 1.2. Описание, постановка задачи и разработка бизнес-правил

Глава 2. Основы теории проектирования баз данных

- 2.1. Информационная модель данных
 - Последовательность создания информационной модели
 - Взаимосвязи в модели
 - Типы моделей данных
- 2.2. Проектирование базы данных
 - Этап 1. Определение сущностей
 - Этап 2. Определение взаимосвязей между сущностями
 - Этап 3. Задание первичных и альтернативных ключей, определение атрибутов сущностей
 - Этап 4. Приведение модели к требуемому уровню нормальной формы
 - Этап 5. Физическое описание модели
- 2.3. Словарь данных
- 2.4. Администрирование базы данных

Глава 3. Обзор возможностей и особенностей различных СУБД

- 3.1. Средства быстрой разработки приложений
- 3.2. Visual FoxPro
 - Project Manager
 - Database Designer
 - Form Designer
 - Visual Class Designer
 - Query / View Designer
 - Connection Designer
 - Report/Label Designer
 - Menu Designer
 - Вспомогательные средства разработчика
- 3.3. Access
 - Запросы
 - Формы
 - Отчеты
 - Макросы
 - Система защиты
- 3.4. Visual Basic
- 3.5. MS SQL Server
- 3.6. Руководство для покупателя

Глава 4. Основы языка программирования

- 4.1. Что такое язык программирования
- 4.2. Как написать программу
- 4.3. "Горячая десятка"

4.4. Еще несколько навязчивых советов

Глава 5. Объектно-ориентированное программирование

5.1. Объектная модель и ее свойства

5.2. Объекты и их свойства

- Объекты для работы с данными
- Объекты для управления работой приложения
- Объекты для оформления интерфейса пользователя
- Объекты-контейнеры
- Невизуальные объекты
- Объекты OLE

5.3. Управление событиями

5.4. Использование методов

Часть 2. Проектирование системы обработки данных

Глава 6. Создание базы данных

6.1. Visual FoxPro

- Создание и модернизация структуры базы данных
- Использование словаря данных
- Создание и модернизация структуры таблиц

6.2. Access

6.3. Visual Basic

6.4. MS SQL Server

- Планирование процесса наращивания

Глава 7. Средства работы с данными

7.1. Организация ввода данных, их поиска и редактирования

- Работа с данными в Visual FoxPro
- Работа с данными в Microsoft Access

7.2. Создание SQL-запросов

- Запросы выборки
- Запросы добавления
- Запросы обновления
- Запросы удаления

7.3. Изменение структуры данных с помощью SQL

7.4. Запросы и локальные представления в Microsoft Visual FoxPro

7.5. Запросы в Microsoft Access

- Запрос добавления
- Запрос - Создание таблицы
- Запрос удаления
- Запрос обновления
- Перекрестный запрос

7.6. Работа с данными в локальной сети

- Visual FoxPro
- Несколько советов по увеличению производительности при работе в сети в приложениях
- MicroxPro
- Microsoft Access

Глава 8. Использование технологии клиент-сервер

8.1. Работа с внешними данными с помощью технологии ODBC

- Команды Transact-SQL

- Создание представлений
- Создание триггеров
- 8.2. Использование Visual FoxPro для разработки клиентского приложения
 - Синхронный и асинхронный процессы
 - Создание внешних представлений
- 8.3. Использование Access и Visual Basic для разработки клиентского приложения
- 8.4. Использование ODBC API для доступа к внешним данным
- 8.5. Remote Data Objects
- 8.6. Внешнее управление сервером с помощью SQL-DMO

Часть 3. Разработка пользовательской программы

Глава 9. Разработка пользовательского интерфейса

- 9.1. Инструментарий разработчика
- 9.2. Конструируем форму
 - Создание формы "Прием заказов" на Visual FoxPro
 - Создание формы "Прием заказов" на Access
- 9.3. Разработка управляющего меню
 - Разработка меню в Visual FoxPro
 - Разработка меню в Access

Глава 10. Использование готовых компонентов в приложении

- 10.1. Основные преимущества модульного проектирования прикладных программ
- 10.2. Как правильно использовать OLE 2.0
 - Возможности OLE 2.0
 - Использование OLE Automation
 - Управление объектами Excel
 - Управление объектами Word for Windows
- 10.3. Использование OLE Automation для передачи данных
 - Построение графиков с помощью MS Graph 5.0
 - Построение графиков с помощью MS Excel 7.0
 - Построение отчета в Word for Windows
 - Запись информации в Schedule+
- 10.4. Применяем ActiveX
 - Иерархический список
 - Календарь

Глава 11. Подготовка отчетных данных

- 11.1. Создание отчетов в Visual FoxPro
 - Управление режимом печати
- 11.2. Создание отчетов в Access

Глава 12. Подготовка и отладка пользовательского приложения

- 12.1. Общие принципы отладки приложения
- 12.2. Инструментальные средства отладки
 - Отладка программы в Visual FoxPro
 - Отладка программы в Access
 - Обработка ошибок процессора баз данных в Access
 - Отладка программы в Visual Basic
- 12.3. Подготовка приложения для распространения

Приложение 1. Дополнительные возможности новой версии Visual FoxPro 5.0

Визуальные средства проектирования
Поставка программного пакета
Требования к установке
Project Manager
Работа с кодом программы
Создание базы данных
Работа с данными
Расширение возможностей технологии клиент-сервер
Построение пользовательского интерфейса
Расширение функций OLE
Отладка приложения

Приложение 2. Взаимозаменяемость команд и функций Visual FoxPro и Visual Basic

Глава 1

Постановка задачи и разработка бизнес-правил

1.1. Некоторые определения

1.2. Описание, постановка задачи и разработка бизнес-правил

Создать хорошее приложение для обработки данных непросто. Это непросто, даже если вы создаете программу "для внутреннего использования", то есть для себя или своих коллег. Создание коммерческого приложения труднее в несколько раз, почему это так, вы поймете после прочтения этой книги. Для начала скажем, что если вы пишете приложение для себя или других пользователей и начали работу с создания файлов и заполнения их данными, считайте, что первая ошибка уже совершена. Опыт показывает, что пока у вас нет четкого и подробного представления о будущем проекте, СУБД - система управления базами данных - вам не нужна.

1.1. Некоторые определения

В черновике рукописи этот параграф назывался "Основные определения теории..." Внимательно следя за первыми рецензентами и неизменно наблюдая резкую смену выражения лиц, авторы, не рассчитывая на большую, чем у рецензентов, выдержку читателей, с помощью клавиши **Del** сильно откорректировали текст. И хотя в заголовке вместо слова "основные" появилось "некоторые", в этом параграфе вы найдете описание терминов, наиболее, на наш взгляд, важных для программиста, который собирается написать приложение для автоматизации обработки данных.

В данном параграфе мы рассмотрим следующие понятия:

- предметная область;
- объект и классы объектов;
- атрибуты и элементы данных;
- значения данных;
- ключевой элемент данных, первичный ключ и альтернативные ключи;
- запись данных;
- тип данных и домены;
- таблица;
- представление;
- связь;

- хранимые процедуры;
- триггеры;
- правила;
- ссылочная целостность;
- нормализация отношений;
- словарь данных.

Каждой цивилизации приходится иметь дело с обработкой информации. С развитием экономики и ростом численности населения возрастает и объем взаимосвязанных данных, необходимых для решения коммерческих и административных задач. Взаимосвязанные данные называют информационной системой. Такая система в первую очередь призвана облегчить труд человека, но для этого она должна как можно лучше соответствовать очень сложной модели реального мира.

Ядром информационной системы являются хранимые в ней данные. На любом предприятии данные различных отделов, как правило, пересекаются, то есть используются в нескольких подразделениях или вообще являются общими. Например, для целей управления часто нужна информация по всему предприятию. Заказ комплектующих невозможен без наличия информации о запасах. Хранящиеся в информационной системе данные должны быть легко доступны в том виде, в каком они нужны для конкретной производственной деятельности предприятия. При этом не имеет существенного значения способ хранения данных. Сегодня на предприятии мы можем встретить систему обработки данных традиционного типа, в которой служащий вручную помещает данные в скоросшиватель, и рядом с ней - современную систему с применением самой быстросействующей ЭВМ, сложнейшего оборудования и программного обеспечения. Несмотря на поразительную несхожесть, обе эти системы обязаны предоставлять достоверную информацию в определенное время, определенному лицу, в определенном месте и с ограниченными затратами.

Чтобы понять процесс построения информационной системы, необходимо знать ряд терминов, которые применяются при описании и представлении данных.

Предметной областью называется часть реальной системы, представляющая интерес для данного исследования.

При проектировании автоматизированных информационных систем предметная область отображается моделями данных нескольких уровней. Число используемых уровней зависит от сложности системы, но в любом случае включает логический и физический уровни. Предметная область может относиться к любому типу организации (например, банк, университет, больница или завод).

Необходимо различать полную предметную область (крупное производственное предприятие, склад, универсам и т.д.) и организационную единицу этой предметной области. Организационная единица в свою очередь может представлять свою предметную область (например, цех по производству кузовов автомобильного завода или отдел обработки данных предприятия по производству ЭВМ). В данном случае цехи и отделы сами могут соответствовать определенным предметным областям.

Информация, необходимая для описания предметной области, зависит от реальной модели и может включать сведения о персонале, заработной плате, товарах, накладных, счетах, отчетах по сбыту, лабораторных тестах, финансовых сделках, историях болезней, то есть сведения о людях, местах, предметах, событиях и понятиях.

Объектом называется элемент информационной системы, информацию о котором мы сохраняем. В реляционной теории баз данных объект называется **сущностью**.

Объект может быть реальным (например, человек, какой-либо предмет или населенный пункт) и абстрактным (например, событие, счет покупателя или изучаемый студентами курс). Так, в области продажи автомобилей примерами объектов могут служить МОДЕЛЬ АВТОМОБИЛЯ, КЛИЕНТ и СЧЕТ. На товарном складе - это ПОСТАВЩИК, ТОВАР, ОТПРАВЛЕНИЕ и т. д. Каждый объект обладает определенным набором свойств, которые запоминаются в информационной системе. При обработке данных часто приходится иметь дело с совокупностью однородных объектов, например таких, как служащие, и записывать информацию об одних и тех же свойствах для каждого из них.

Классом объектов называют совокупность объектов, обладающих одинаковым набором свойств.

Таким образом, для объектов одного класса набор свойств будет одинаков, хотя значения этих свойств для каждого объекта, конечно, могут быть разными. Например, класс объектов МОДЕЛЬ АВТОМОБИЛЯ будет иметь одинаковый набор свойств, описывающих характеристики автомобилей, и каждая модель будет иметь различные значения этих характеристик.

Объекты и их свойства являются понятиями реального мира. В мире информации, существующем в представлении программиста, говорят об атрибутах объектов.

Атрибут - это информационное отображение свойств объекта. Каждый объект характеризуется рядом основных атрибутов.

Например, модель автомобиля характеризуется типом кузова, рабочим объемом двигателя, количеством цилиндров, мощностью, габаритами, названием и т. д. Клиент магазина, продающего автомобили, имеет такие атрибуты, как фамилию, имя, отчество, адрес и, возможно, идентификационный номер. Каждый атрибут в модели должен иметь уникальное имя - идентификатор. Атрибут при реализации информационной модели на каком-либо носителе информации часто называют элементом данных, полем данных или просто полем. Взаимосвязь между перечисленными выше понятиями проиллюстрирована схемой, приведенной на рис. 1.1.



Рис. 1.1. Три области представления данных

Таблица - это некоторая регулярная структура, состоящая из конечного набора однотипных записей. В некоторых источниках таблица называется отношением.

Мы постараемся избегать последнего термина, так как с развитием реляционной теории "отношением" наряду с термином "связь" часто стали называть связи между таблицами. Каждая запись одной таблицы состоит из конечного (и одинакового!) числа полей, причем конкретное поле каждой записи одной таблицы может содержать данные только одного типа.

Значение данных представляет собой действительные данные, содержащиеся в каждом элементе данных.

Элемент данных "НАИМЕНОВАНИЕ МОДЕЛИ" может принимать такие значения, как "Voyager'96 3.8 Grand", "Continental 4.6" или "Crown Victoria 4.6". В зависимости от того, как элементы данных описывают объект, их значения могут быть количественными, качественными или описательными.

Информацию о некоторой предметной области можно представить с помощью нескольких объектов, каждый из которых описывается несколькими элементами данных. Принимаемые элементами данных значения называются данными. Единичный набор принимаемых элементами данных значений называется экземпляром объекта. Объекты связываются между собой определенным образом. Соответствующая модель объектов с составляющими их элементами

данных и взаимосвязями называется **концептуальной моделью**. Концептуальная модель дает общее представление о потоке данных в предметной области.

Некоторые элементы данных обладают важным для построения информационной модели свойством. Если известно значение, которое принимает такой элемент данных объекта, мы можем идентифицировать значения, которые принимают другие элементы данных этого же объекта. Например, зная уникальный номер модели автомобиля - 7, мы можем определить, что это "Voyager'96 3.8 Grand" и что рабочий объем двигателя у данной модели "3778".

Ключевым элементом данных называется такой элемент, по которому можно определить значения других элементов данных.

Однозначно идентифицировать объект могут два и более элемента данных. В этом случае их называют "кандидатами" в ключевые элементы данных. Вопрос о том, какой из кандидатов использовать для доступа к объекту, решается пользователем или разработчиком системы. Выбирать ключевые элементы данных следует тщательно, поскольку правильный выбор способствует созданию достоверной концептуальной модели данных.

Первичный ключ - это атрибут (или группа атрибутов), которые единственным образом идентифицируют каждую строку в таблице.

Понятие первичного ключа является исключительно важным в связи с понятием целостности баз данных, которое мы подробно рассмотрим в конце этого параграфа.

Альтернативный ключ - это атрибут (или группа атрибутов), несовпадающий с первичным ключом и уникально идентифицирующий экземпляр объекта.

Например, для объекта "служащий", который имеет атрибуты "ИДЕНТИФИКАТОР СЛУЖАЩЕГО", "ФАМИЛИЯ", "ИМЯ" и "ОТЧЕСТВО", группа атрибутов "ФАМИЛИЯ", "ИМЯ", "ОТЧЕСТВО" может являться альтернативным ключом по отношению к атрибуту "ИДЕНТИФИКАТОР СЛУЖАЩЕГО" (в предположении, что на предприятии не работают полные тезки).

Запись данных - это совокупность значений связанных элементов данных.

На рис. 1.2 такими элементами данных являются уникальный ключ и наименование модели, рабочий объем, количество цилиндров и мощность двигателя. Например, одна из записей - "7 Voyager'96 3.8 Grand 3778 6 164,0". Эта строка представляет собой значения, которые принимают элементы данных объекта МОДЕЛЬ АВТОМОБИЛЯ (MODEL). Такие строки формируют записи данных. Записи хранятся на некотором носителе, в качестве которого может выступать человеческий мозг, лист бумаги, память ЭВМ, внешнее запоминающее устройство и т. д.

Уникальный ключ модели	Наименование модели	Рабочий объем (куб см)	Мощность (лс)
1	145 1.4	1351	90,0
2	146 1.9	1929	90,0
3	740i 4.0	3982	296,0
4	840Ci 4.0	3982	296,0
5	M3 3.0	3201	321,0
6	GMC Jimmy 4.3	4300	193,0
7	Voyager'96 3.8 Grand	3778	164,0
8	ZX 2.0	1998	150,0
9	Stealth 3.0	2972	166,0
10	348 Spider 3.4	3405	320,0

Рис. 1.2. Записи данных объекта MODEL

Тип данных характеризует вид хранящихся данных.

Понятие типа данных в информационной модели полностью адекватно понятию типа данных в языках программирования. Обычно в современных СУБД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (например, суммы в денежных единицах), а также данных специального формата (дата, время, временной интервал и пр.). В любом случае при выборе типа данных следует учитывать возможности той СУБД, с помощью которой будет реализовываться физическая модель информационной системы.

Доменом называется набор значений элементов данных одного типа, отвечающий поставленным условиям.

Понятие домена более специфично для баз данных, хотя и имеет определенные аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных, который "забраковывает" недопустимые значения. Если вычисление этого логического выражения дает результат "истина", то элемент данных является элементом домена. Например, домен "НАИМЕНОВАНИЕ" в нашем примере (рис. 1.3) определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, очевидно, что такие строки не должны начинаться с какого-нибудь специфичного символа типа знака подчеркивания или тире). В упрощенном виде понятие домена может характеризоваться как потенциальное множество допустимых значений одного типа. Например, значением атрибута "ПОЛ" может быть только либо "мужской", либо "женский". Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов "КЛЮЧ МОДЕЛИ" и "РАБОЧИЙ ОБЪЕМ" относятся к типу целых чисел, но не являются сравнимыми.

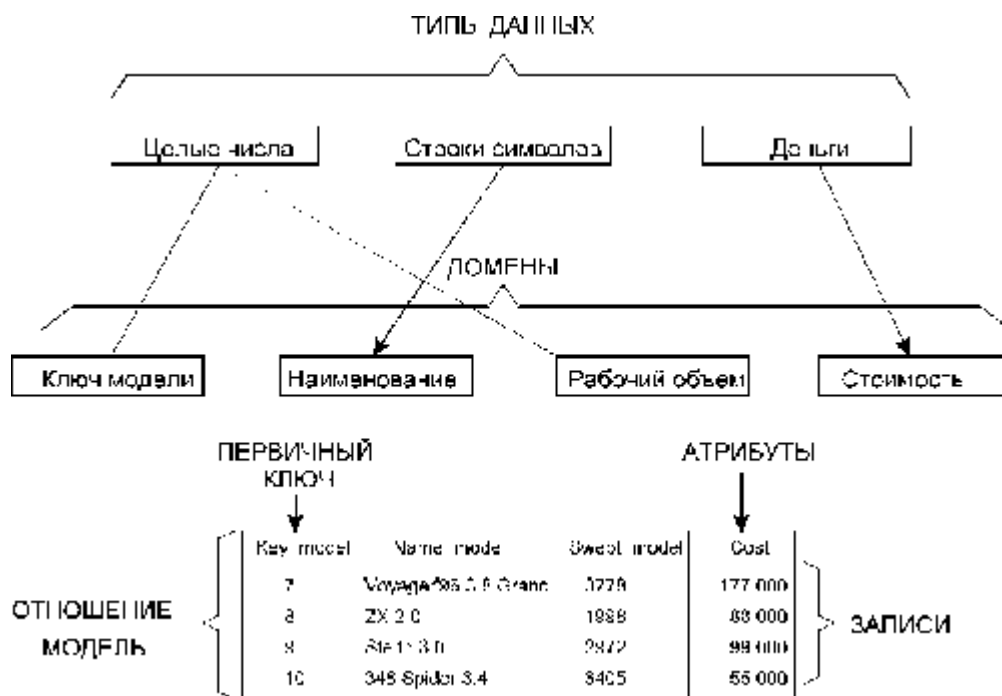


Рис. 1.3. Иерархия понятий в таблице МОДЕЛЬ

Представление - это сохраняемый в базе данных именованный запрос на выборку данных (из одной или нескольких таблиц).

Результатом выполнения любого запроса на выборку данных является таблица, и поэтому концептуально можно относиться к любому представлению как к таблице.

Связь - это функциональная зависимость между сущностями.

Если между некоторыми сущностями существует связь, то факты из одной сущности ссылаются или некоторым образом связаны с фактами из другой сущности. Поддержание непротиворечивости функциональных зависимостей между сущностями называется ссылочной целостностью. Поскольку связи содержатся "внутри" реляционной модели, реализация ссылочной целостности может выполняться как приложением, так и самой СУБД (с помощью механизмов декларативной ссылочной целостности и триггеров).

Связи могут быть представлены пятью основными характеристиками:

- тип связи (идентифицирующая, не идентифицирующая, полная/неполная категория, неспецифическая связь);
- родительская сущность;
- дочерняя (зависимая) сущность;
- мощность связи (*cardinality*);
- допустимость пустых (*null*) значений.

Связь называется идентифицирующей, если экземпляр дочерней сущности идентифицируется (однозначно определяется) через ее связь с родительской сущностью. Атрибуты, составляющие первичный ключ родительской сущности, при этом входят в первичный ключ дочерней сущности. Дочерняя сущность при идентифицирующей связи всегда является зависимой.

Связь называется не идентифицирующей, если экземпляр дочерней сущности идентифицируется иначе, чем через связь с родительской сущностью. Атрибуты, составляющие первичный ключ родительской сущности, при этом входят в состав не ключевых атрибутов дочерней сущности.

Мощность связи представляет собой отношение количества экземпляров родительской сущности к соответствующему количеству экземпляров дочерней сущности. Для любой связи, кроме неспецифической, эта связь записывается как 1:n.

Хранимые процедуры - это приложение (программа), объединяющее запросы и процедурную логику (операторы присваивания, логического ветвления и т. д.) и хранящееся в базе данных.

Хранимые процедуры позволяют содержать вместе с базой данных достаточно сложные программы, выполняющие большой объем работы без передачи данных по сети и взаимодействия с клиентом. Как правило, программы, записываемые в хранимых процедурах, связаны с обработкой данных. Тем самым база данных может представлять собой функционально самостоятельный уровень приложения, который может взаимодействовать с другими уровнями для получения запросов или обновления данных.

Правила позволяют вызывать выполнение заданных действий при изменении или добавлении данных в базу данных (БД) и тем самым контролировать истинность помещаемых в нее данных.

Обычно действие - это вызов определенной процедуры или функции. Правила могут ассоциироваться с полем или записью и, соответственно, срабатывать при изменении данных в конкретном поле или записи таблицы. Нельзя использовать правила при удалении данных.

В отличие от ограничений, которые являются лишь средством контроля относительно простых условий корректности ввода данных, правила позволяют проверять и поддерживать сколь угодно сложные соотношения между элементами данных в БД.

Триггеры - это предварительно определенное действие или последовательность действий, автоматически осуществляемых при выполнении операций обновления, добавления или удаления данных.

Триггер является мощным инструментом контроля за изменением данных в БД, а также помогает программисту автоматизировать операции, которые должны выполняться в этом случае. Триггер выполняется после проверки правил обновления данных.

Обратите внимание на исключительную важность в этом определении слова "автоматически".

Ни пользователь, ни приложение не могут активизировать триггер, он выполняется автоматически, когда пользователь или приложение выполняют с БД определенные действия. Триггер включает в себя следующие компоненты:

- Ограничения, для реализации которых собственно и создается триггер.
- Событие, которое будет характеризовать возникновение ситуации, требующей проверки ограничений. События чаще всего связаны с изменением состояния БД (например, добавление записи в какую-либо таблицу), но могут учитываться и дополнительные условия (например, добавление записи только с отрицательным значением).
- Предусмотренное действие выполняется за счет выполнения процедуры или последовательности процедур, с помощью которых реализуется логика, требуемая для реализации ограничений.

Использование триггеров при проектировании БД позволяет получить при разработке приложения следующие преимущества:

- Триггеры всегда выполняются при совершении соответствующих действий. Разработчик продумывает использование триггеров при проектировании БД и может больше не вспоминать о них при разработке приложения для доступа к данным. Если для работы с этой же БД вы решите создать новое приложение, триггеры и там будут отрабатывать заданные ограничения.
- При необходимости триггеры можно изменять централизованно непосредственно в БД. Пользовательские программы, использующие данные из этой БД, не потребуют модернизации.
- Система обработки данных, использующая триггеры, обладает лучшей переносимостью в архитектуру клиент-сервер за счет меньшего объема требуемых модификаций.

Ссылочная целостность - это обеспечение соответствия значения внешнего ключа экземпляра дочерней сущности значениям первичного ключа в родительской сущности. Ссылочная целостность может контролироваться при всех операциях, изменяющих данные.

Для каждой связи на логическом уровне могут быть заданы требования по обработке операций добавления, обновления или удаления данных для родительской и дочерней сущности. Могут использоваться следующие варианты обработки этих событий:

- отсутствие проверки;
- проверка допустимости;
- запрет операции;
- каскадное выполнение операции обновления или удаления данных сразу в нескольких связанных таблицах;
- установка пустого (NULL) значения или заданного значения по умолчанию.

Нормализация отношений - это процесс построения оптимальной структуры таблиц и связей в реляционной БД.

В процессе нормализации элементы данных группируются в таблицы, представляющие объекты и их взаимосвязи. Теория нормализации основана на том, что определенный набор таблиц обладает лучшими свойствами при включении, модификации и удалении данных, чем все остальные наборы таблиц, с помощью которых могут быть представлены те же данные.

Словарь данных - это централизованное хранилище сведений об объектах, составляющих их элементах данных, взаимосвязях между объектами, их источниках, значениях, использовании и форматах представления.

В [следующей главе](#) мы будем подробно говорить о словаре данных и рассмотрим практический пример его использования.

1.2. Описание, постановка задачи и разработка бизнес-правил

Занимаясь разработкой автоматизированных систем на протяжении длительного времени, мы можем отметить, что в 77 случаях из 100 заказчик сам не знает, чего хочет, и в 99 из 100 постановку задачи приходится воспринимать на слух, в процессе работы неоднократно уточняя те или иные мелкие, но значимые моменты будущего приложения. Более того, в таких случаях очень часто, при очередном обсуждении с заказчиком возможностей новой системы, мы открываем для себя все новые и новые горизонты предстоящей работы, требующие существенного расширения функциональности будущей программы. Но это не самый плохой вариант. Иной раз уже через несколько часов после прощального рукопожатия заказчик полностью переосмысливает свои цели, после чего задача в корне меняется, и следующий визит заставляет нас начать всю работу заново. Именно по этой причине мы рекомендуем вам, внимательно выслушав заказчика, попросить его описать задачу в письменном виде, на основании чего самостоятельно сформулировать постановку задачи. Уверяем вас, если результат окажется положительным, то это будет признанием того, что ваш заказчик действительно нуждается в данной программе, а самое главное, знает чего хочет.

В этом параграфе мы попробуем уяснить, как должно выглядеть описание и постановка задачи.

Некая фирма "Фронтон" (название придумали большие любители технологии клиент-сервер, поэтому его надо читать на зарубежный манер: вдохните и скажите "фронт", сделайте задержку дыхания и на выдохе произнесите "он") занимается продажей легковых автомобилей на заказ. Процесс продажи протекает следующим образом. Покупатель производит заказ на покупку автомобиля, пользуясь предоставленным ему фирмой каталогом легковых автомобилей. Представитель фирмы выписывает счет на выбранную модель автомобиля и одновременно с этим отправляет запрос о приобретении данного автомобиля на завод-изготовитель (фирме - поставщику). Фирма "Фронтон" заключила юридические соглашения о поставке автомобилей с рядом заводов-изготовителей легковых автомобилей и крупных дистрибьюторов. После оплаты по соответствующему счету фирма "Фронтон" подтверждает заказ о приобретении и обязуется в течение четырехнедельного периода предоставить покупку соответствующему покупателю.

В максимально простом виде схема бизнес-процесса фирмы "Фронтон" представлена на рис. 1.4. На основании исследований рынка потенциальных покупателей и предложений автоиндустрии служба или отдельный специалист разрабатывает каталог предлагаемых к продаже автомобилей; в большой фирме такую службу называли бы отделом маркетинга. Каталог распространяется на рынке потенциальных покупателей. С клиентом, решившим приобрести автомобиль, работает служба оформления заказов. Специалисты, входящие в эту службу, принимают заказ, уточняют комплектацию выбранного автомобиля, отправляют счета, следят за их оплатой и наконец вручают клиенту поступивший автомобиль. Служба внутренней поддержки обеспечивает распределение работы по исполнителям и решает возникающие проблемы, например, ограничения доступа к данным.

При анализе бизнес-процесса фирмы полезно ответить на 6 вопросов: что, как, где, кто, когда и почему.

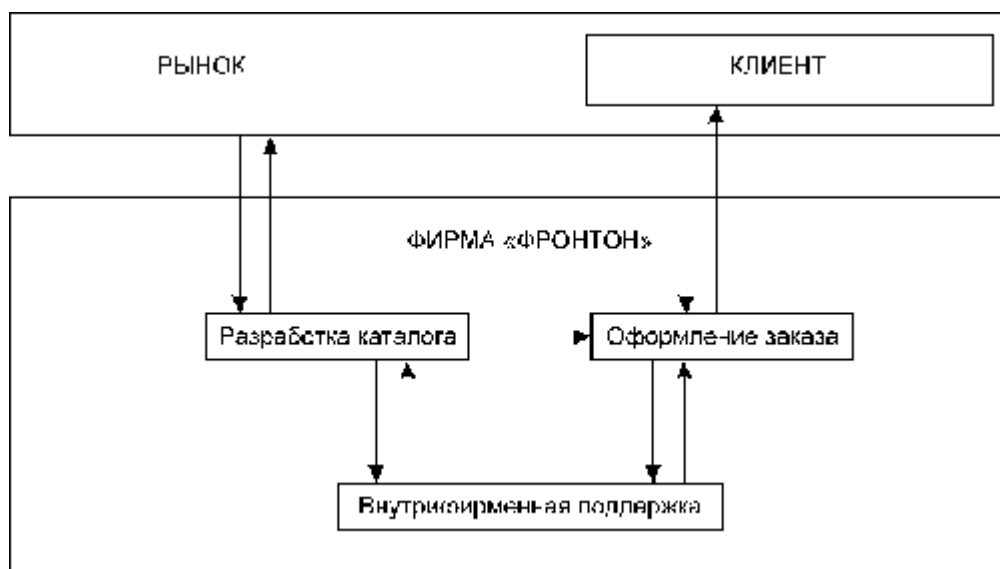


Рис. 1.4. Упрощенная схема бизнес-процесса

При ответе на первый вопрос: "Что лежит в основе бизнеса данной фирмы?", как правило, выявляются наиболее важные для данного бизнеса или производственного процесса компоненты. В нашем случае это будут:

- сотрудники;
- клиенты (покупатели);
- поставщики;
- каталог;
- автомобили;
- заказы.

Ответы на второй вопрос: "Как это делается?" позволяют получить список основных бизнес-процессов, происходящих в фирме. Для нашего примера в такой список можно включить следующие пункты:

- составление каталога;
- рассылка каталога;
- анализ рынка;
- продажи;
- оформление счетов и накладных;
- управление работой персонала;
- реклама;
- решение бухгалтерских задач.

Вопрос: "Где происходят данные процессы?" больше относится к проблемам телекоммуникаций и организации совместной работы персонала. Ведь в случае, например, большого объема операций, которые выполняются вне территории фирмы торговыми агентами, придется учитывать проблемы синхронизации данных. При наличии филиалов весьма непростой проблемой является оптимальный выбор системы распределения данных. Можно централизовать всю обработку данных, и филиалы будут выполнять свои операции, пользуясь возможностями телекоммуникаций. Работа с данными в этом случае упрощается, но каково будет удивление клиента, когда вы ему сообщите, что не можете взять у него несколько десятков тысяч долларов и продать приглянувшуюся машину, так как оборвалась связь с центральным офисом. В рассматриваемом примере допустим, что все операции выполняются в пределах одного здания, а организация совместного использования данных основана на возможностях локальной сети и сервера БД.

Ответ на вопрос: "Кто выполняет эти процессы?" даст организационная структура фирмы. Упрощенная организационная структура фирмы "Фронтон" представлена на рис. 1.5.

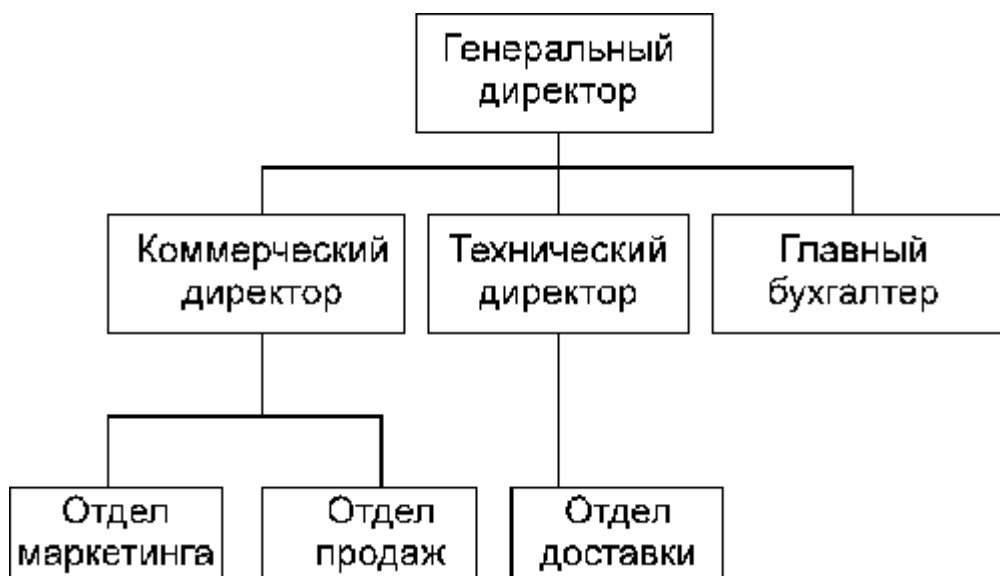


Рис. 1.5.

Важно получить и ответ на вопрос: "Когда выполняется то или иное действие?" Это прояснит периодичность осуществляемых бизнес-процессов и позволит правильно расставить акценты в

будущей прикладной программе. В нашем случае примем такую временную последовательность выполняемых процессов:

- обновление каталога - раз в год и внесение поправок в экстренных случаях;
- подведение итогов продаж - ежемесячно;
- годовой отчет - ежегодно к 20 февраля.

Последний вопрос: "Почему эти действия выполняются?" позволяет определить мотивацию производственной деятельности фирмы. Бизнес-задачи фирмы "Фронтон" определим так:

- достижение наилучшего соотношения "затраты - удобство" для клиентов;
- обеспечение условий для успешной деятельности персонала;
- получение приемлемой прибыли;
- повышение доходов при автоматизации обработки данных.

Ответы на шесть перечисленных вопросов позволяют подойти к главному в постановке задачи - построению информационной модели предприятия. В простейшем виде такая модель может быть отображена в виде взаимосвязей между бизнес-компонентами и бизнес-процессами, как это показано на рис. 1.6. В практике проектирования информационных систем такие схемы получили название **ER-диаграмм** (Entity-relationship diagram (ERD) - диаграмма "Сущность-связь"). ER-диаграммы хорошо вписываются в методологию структурного анализа и проектирования информационных систем. Такие методологии обеспечивают строгое и наглядное описание проектируемой системы, которое начинается с ее общего обзора и затем уточняется, давая возможность получить различную степень детализации объекта с различным числом уровней.

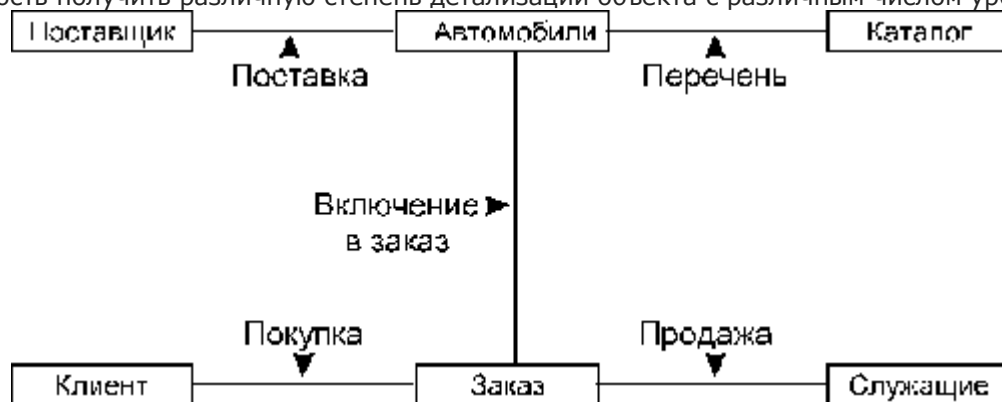


Рис. 1.6. Диаграмма взаимосвязей между бизнес-компонентами и бизнес-процессами

Максимально формализованное описание задачи в нашем примере будет выглядеть следующим образом.

Наименование задачи:

Автоматизация управления работой дилера по продаже легковых автомобилей "AUTO STORE".

Цель работы дилера:

Продажа легковых автомобилей на заказ по каталогу.

Функции дилера:

- Заключение договоров на поставку автомобилей.
- Ведение каталога автомобилей, предлагаемых на продажу.
- Прием заказов у клиентов (покупателей) на поставку автомобилей.
- Работа с клиентами (маркетинг): реклама новых автомобилей, подготовка сведений о приобретаемых автомобилях, анализ продаж, ведение справочника клиентов.
- Отправка заказов поставщику автомобилей.
- Ведение расчетов за проданные автомобили (выписка накладных).
- Учет валютного курса.

Бизнес-правила:

- Сведения о клиентах хранятся 10 лет.
- Оплата ожидается 3 недели, если ее не происходит, заказ уничтожается.

- Подтверждение запроса о приобретении автомобиля отправляется фирме-поставщику после прихода денег.
- При отказе от поставленного автомобиля с покупателя удерживается 9% суммы оплаты по счету, данная величина должна регулироваться.
- Срок поставки 4 недели после прихода денег.
- Просрочка доставки автомобиля клиенту оплачивается фирмой "Фронтон" из расчета 0,1% в день, данная величина должна регулироваться.
- Если автомобиль не поставлен в течение 2 месяцев, возвращается вся сумма оплаты и пеня.

Требования к программе:

Программа должна работать под управлением операционных систем Windows 95 или Windows NT.

Перечень вводимой информации:

- наименование модели продаваемого автомобиля;
- рабочий объем двигателя, см³;
- количество цилиндров в двигателе;
- номинальная мощность двигателя, л.с.;
- максимальный крутящий момент, НФм;
- максимальная скорость автомобиля, км/ч;
- время разгона автомобиля до 100 км/ч, с;
- количество дверей;
- количество мест;
- длина, мм;
- ширина, мм;
- высота, мм;
- расход топлива при скорости 90 км/ч, л/100 км;
- расход топлива при скорости 120 км/ч, л/100 км;
- расход топлива при городском цикле, л/100 км;
- наименование производителя автомобиля;
- наименование страны, в которой производится автомобиль;
- наименование используемого автомобилем топлива;
- наименование шин;
- наименование типа кузова;
- дата выпуска автомобиля;
- стоимость автомобиля;
- наименование клиента;
- адрес клиента;
- телефон клиента;
- факс клиента;
- фамилия, имя и отчество клиента;
- признак юридического лица клиента;
- примечание для записи заметок по работе с клиентом;
- номер счета;
- дата продажи;
- сумма продажи;
- пометка об оплате;
- фамилия, имя и отчество продавца.

Перечень печатных отчетов:

- номенклатура предлагаемых к продаже автомобилей;
- список клиентов;
- анализ продаж;
- список заказов;
- счет на покупку.

Требования к оснащению офиса фирмы компьютерной техникой:

- Для пользователей: ПЭВМ не ниже Pentium 100/16/420 с операционной системой Windows 95 или Windows NT Workstation и пакетом программ MS Office.
- Сервер не ниже Pentium 166/32/1000 с операционной системой Windows NT Server и MS SQL Server 6.x.
- Локальная сеть.
- Сетевой лазерный или струйный принтер.

Глава 2

Основы теории проектирования баз данных

2.1. Информационная модель данных

Последовательность создания информационной модели

Взаимосвязи в модели

Типы моделей данных

2.2. Проектирование базы данных

Этап 1. Определение сущностей

Этап 2. Определение взаимосвязей между сущностями

Этап 3. Задание первичных и альтернативных ключей, определение атрибутов сущностей

Этап 4. Приведение модели к требуемому уровню нормальной формы

Этап 5. Физическое описание модели

2.3. Словарь данных

2.4. Администрирование базы данных

Если вам удалось осмыслить задачу, поставленную перед вами заказчиком, примерно так, как было описано в [предыдущей главе](#), вы на правильном пути. Но это не означает, что наконец-то настала пора воплощать задуманное в "материализованном" виде. Необходимо потратить еще немного усилий для приведения полученной от заказчика информации к наиболее подходящему виду.

2.1. Информационная модель данных

При проектировании системы обработки данных именно данные и интересуют нас в первую очередь. Причем больше всего нас интересует организация данных. Помочь понять организацию данных призвана информационная модель.

В этом параграфе мы познакомимся:

- с общими принципами разработки информационной модели;
- с отличиями между концептуальной, логической и физической моделями данных;
- с различными видами взаимосвязей между элементами модели.

Система автоматизированной обработки данных основывается на использовании определенной модели данных или информационной модели. Модель данных отражает взаимосвязи между объектами.

Последовательность создания информационной модели

Процесс создания информационной модели начинается с определения концептуальных требований ряда пользователей (рис. 2.1). Концептуальные требования могут определяться и для некоторых задач (приложений), которые в ближайшее время реализовывать не планируется. Это может несколько повысить трудоемкость работы, однако поможет наиболее полно учесть все нюансы функциональности, требуемой для разрабатываемой системы, и снизит вероятность ее переделки в дальнейшем. Требования отдельных пользователей интегрируются в едином "обобщенном представлении". Последнее называют концептуальной моделью.

Концептуальная модель представляет объекты и их взаимосвязи без указания способов их физического хранения.

Таким образом, концептуальная модель является, по существу, моделью предметной области. При проектировании концептуальной модели все усилия разработчика должны быть направлены в основном на структуризацию данных и выявление взаимосвязей между ними без рассмотрения особенностей реализации и вопросов эффективности обработки. Проектирование концептуальной модели основано на анализе решаемых на этом предприятии задач по обработке данных. Концептуальная модель включает описания объектов и их взаимосвязей, представляющих интерес в рассматриваемой предметной области и выявляемых в результате анализа данных. Здесь имеются в виду данные, используемые как в уже разработанных прикладных программах, так и в тех, которые только будут реализованы.

Концептуальная модель транслируется затем в модель данных, совместимую с выбранной СУБД. Возможно, что отраженные в концептуальной модели взаимосвязи между объектами окажутся впоследствии нереализуемыми средствами выбранной СУБД. Это потребует изменения концептуальной модели. Версия концептуальной модели, которая может быть обеспечена конкретной СУБД, называется логической моделью.

Логическая модель отражает логические связи между элементами данных вне зависимости от их содержания и среде хранения.

Логическая модель данных может быть реляционной, иерархической или сетевой. Пользователям выделяются подмножества этой логической модели, называемые внешними моделями (в некоторых источниках их также называют подсхемами), отражающие их представления о предметной области. Внешняя модель соответствует представлениям, которые пользователи получают на основе логической модели, в то время как концептуальные требования отражают представления, которые пользователи первоначально желали иметь и которые легли в основу разработки концептуальной модели. Логическая модель отображается в физическую память, такую, как диск, лента или какой-либо другой носитель информации.

Физическая модель, определяющая размещение данных, методы доступа и технику индексирования, называется внутренней моделью системы.

Внешние модели никак не связаны с типом физической памяти, в которой будут храниться данные, и с методами доступа к этим данным. Это положение отражает первый уровень независимости данных. С другой стороны, если концептуальная модель способна учитывать расширение требований к системе в будущем, то вносимые в нее изменения не должны оказывать влияния на существующие внешние модели. Это - второй уровень независимости данных. Уровни независимости данных показаны на рис. 2.1. Важно помнить, что построение логической модели обусловлено требованиями используемой СУБД. Поэтому при замене СУБД она также может измениться.

С точки зрения прикладного программирования независимость данных определяется не техникой программирования, а его дисциплиной. Например, для того чтобы при любом изменении системы избежать перекомпиляции приложения, рекомендуется не определять константы (постоянные значения данных) в программе. Лучшее решение состоит в передаче программе значений в качестве параметров.

Все актуальные требования предметной области и адекватные им "скрытые" требования на стадии проектирования должны найти свое отражение в концептуальной модели. Конечно, нельзя предусмотреть все возможные варианты использования и изменения базы данных. Но в большинстве предметных областей такие основные данные, как объекты и их взаимосвязи, относительно стабильны. Меняются только информационные требования, то есть способы использования данных для получения информации.

Степень независимости данных определяется тщательностью проектирования базы данных. Всесторонний анализ объектов предметной области и их взаимосвязей минимизирует влияние изменений требований к данным в одной программе на другие программы. В этом и состоит всеобъемлющая независимость данных.

Основное различие между указанными выше тремя типами моделей данных (концептуальной, логической и физической) состоит в способах представления взаимосвязей между объектами. При проектировании БД нам потребуется различать взаимосвязи между объектами, между атрибутами одного объекта и между атрибутами различных объектов.

Взаимосвязи в модели

Взаимосвязь выражает отображение или связь между двумя множествами данных. Различают взаимосвязи типа "один к одному", "один ко многим" и "многие ко многим".

В рассматриваемой задаче по автоматизации управления работой дилера по продаже легковых автомобилей, если клиент производит заказ на покупку автомобиля впервые, осуществляется первичная регистрация его данных и сведений о сделанном заказе. Если же клиент производит заказ повторно, осуществляется регистрация только данного заказа. Вне зависимости от того, сколько раз данный клиент производил заказы, он имеет уникальный идентификационный номер (уникальный ключ клиента). Информация о каждом клиенте включает наименование клиента, адрес, телефон, факс, фамилию, имя, отчество, признак юридического лица и примечание. Таким образом, атрибутами объекта КЛИЕНТ являются "УНИКАЛЬНЫЙ КЛЮЧ КЛИЕНТА", "НАИМЕНОВАНИЕ КЛИЕНТА", "АДРЕС КЛИЕНТА" и т. д.

Следующий представляющий для нас интерес объект - МОДЕЛЬ АВТОМОБИЛЯ. Этот объект имеет атрибуты "УНИКАЛЬНЫЙ КЛЮЧ МОДЕЛИ", "НАИМЕНОВАНИЕ МОДЕЛИ" и т. д.

Третий рассматриваемый объект - ЗАКАЗ. Его атрибутами являются "НОМЕР ЗАКАЗА", "КЛЮЧ КЛИЕНТА" и "КЛЮЧ МОДЕЛИ".

И четвертый рассматриваемый объект - ПРОДАВЕЦ. Его атрибутами являются "УНИКАЛЬНЫЙ КЛЮЧ ПРОДАВЦА", "ИМЯ ПРОДАВЦА", "ФАМИЛИЯ" и "ОТЧЕСТВО".

Взаимосвязь "один к одному" (между двумя типами объектов)

Мысленно вернемся к временам планово-распределительной экономики. Допустим, в определенный момент времени один клиент может сделать только один заказ. В этом случае между объектами КЛИЕНТ и ЗАКАЗ устанавливается взаимосвязь "один к одному", обозначаемая одинарными стрелками, как это показано на рис. 2.2,а.

Между данными, хранящимися в объектах КЛИЕНТ и ЗАКАЗ, будет существовать взаимосвязь, в которой каждая запись в одном объекте будет однозначно указывать на запись в другом объекте. На рис. 2.3 приведен пример такой взаимосвязи между данными. Ни в одном, ни в другом объекте не может существовать записи, не связанной с какой-либо записью в другом объекте.

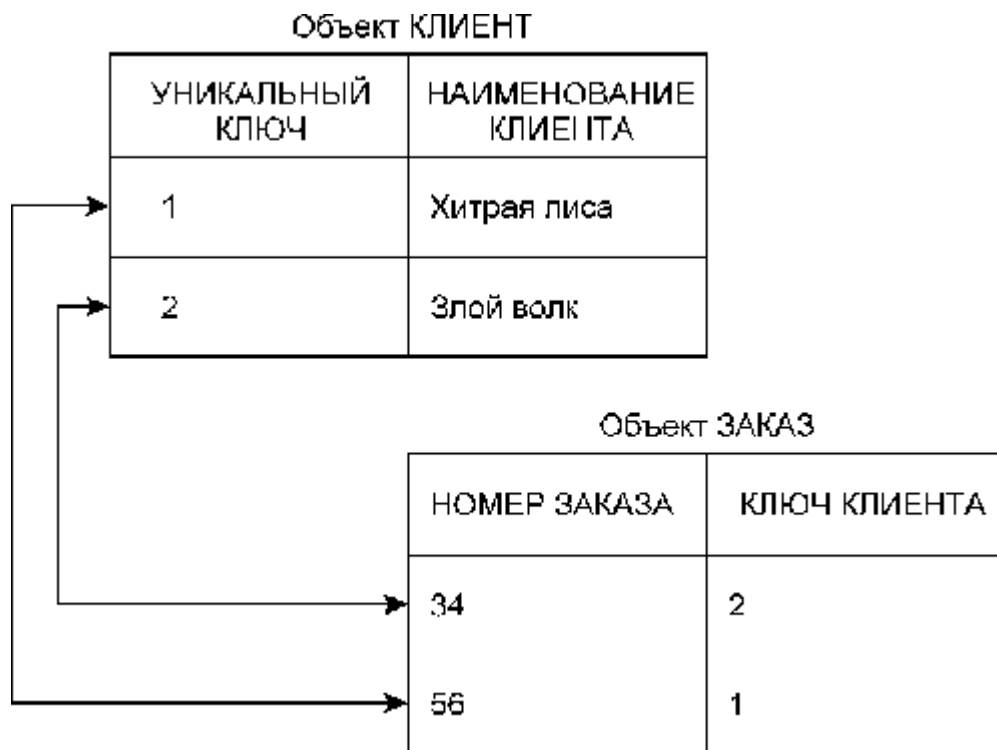


Рис. 2.3. Взаимосвязь между данными при отношении "один к одному"

Взаимосвязь "один ко многим" (между двумя типами объектов)

В определенный момент времени один клиент может стать обладателем нескольких моделей автомобилей, при этом несколько клиентов не могут являться обладателями одного автомобиля. Взаимосвязь "один ко многим" можно обозначить с помощью одинарной стрелки в направлении к "одному" и двойной стрелки в направлении ко "многим", как это показано на рис. 2.2,б.

В этом случае одной записи данных первого объекта (его часто называют родительским или основным) будет соответствовать несколько записей второго объекта (дочерного или подчиненного). Взаимосвязь "один ко многим" очень распространена при разработке реляционных баз данных. В качестве родительского объекта часто выступает справочник, а в дочернем хранятся уникальные ключи для доступа к записям справочника. В нашем примере в качестве такого справочника можно представить объект КЛИЕНТ, в котором хранятся сведения о всех клиентах. При обращении к записи для определенного клиента нам доступен список всех

покупок, которые он сделал и сведения о которых хранятся в объекте МОДЕЛЬ АВТОМОБИЛЯ, как это показано на рис. 2.4. В случае, если в дочернем объекте будут какие-то записи, для которых нет соответствующих записей в объекте КЛИЕНТ, то мы их не увидим. В этом случае говорят, что объект содержит потерянные (одинокие) записи. Это не допустимо, и в дальнейшем вы узнаете, как избегать подобных ситуаций.

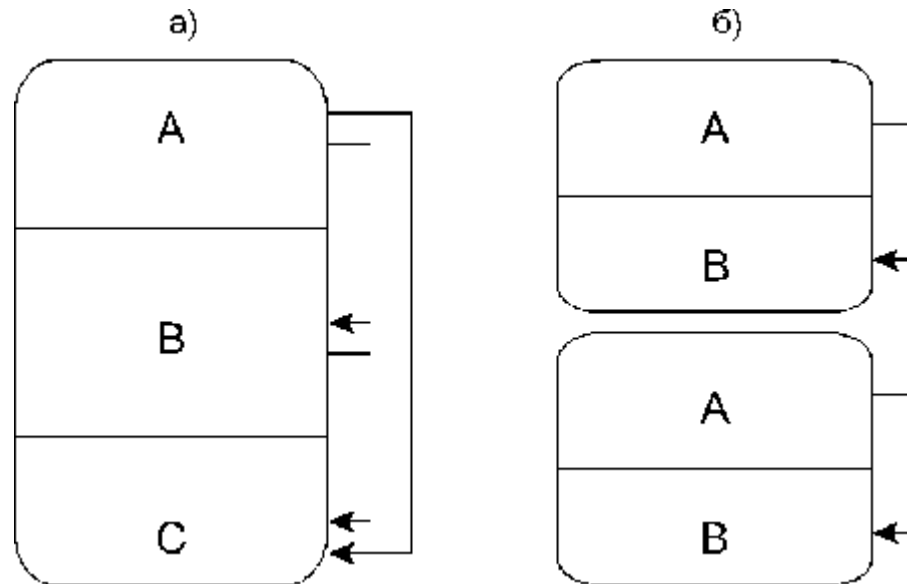


Рис. 2.4. Взаимосвязь между данными при отношении "один ко многим"

Если мы будем просматривать записи объекта МОДЕЛЬ АВТОМОБИЛЯ, то в объекте КЛИЕНТ мы сможем получить данные о клиенте, купившем данный автомобиль (см. рис. 2.4). Обратите внимание, что для потерянных записей сведений о клиенте мы не получим.

Взаимосвязь "многие ко многим" (между двумя типами объектов)

В рассматриваемом примере каждый продавец может обслуживать нескольких клиентов. С другой стороны, приобретая автомобили в различное время, каждый клиент вполне может быть обслужен различными продавцами. Между объектами КЛИЕНТ и ПРОДАВЕЦ существует взаимосвязь "многие ко многим". Такая взаимосвязь обозначается двойными стрелками, как это показано на рис. 2.2, в. На рис. 2.5 приведена схема, по которой в этом случае будут взаимосвязаны данные. При просмотре данных в объекте КЛИЕНТ мы сможем узнать, какие продавцы обслуживали определенного клиента. Однако в объекте ПРОДАВЕЦ в этом случае нам придется завести несколько записей для каждого продавца. Каждая строчка будет соответствовать каждому обслуживанию продавцом клиента. При таком подходе мы столкнемся с серьезными проблемами. Например, не сможем ввести в объект ПРОДАВЕЦ уникальный ключ для каждого продавца, так как неизбежно один продавец будет обслуживать нескольких клиентов, и в этом случае у нас появится несколько записей для одного и того же продавца.

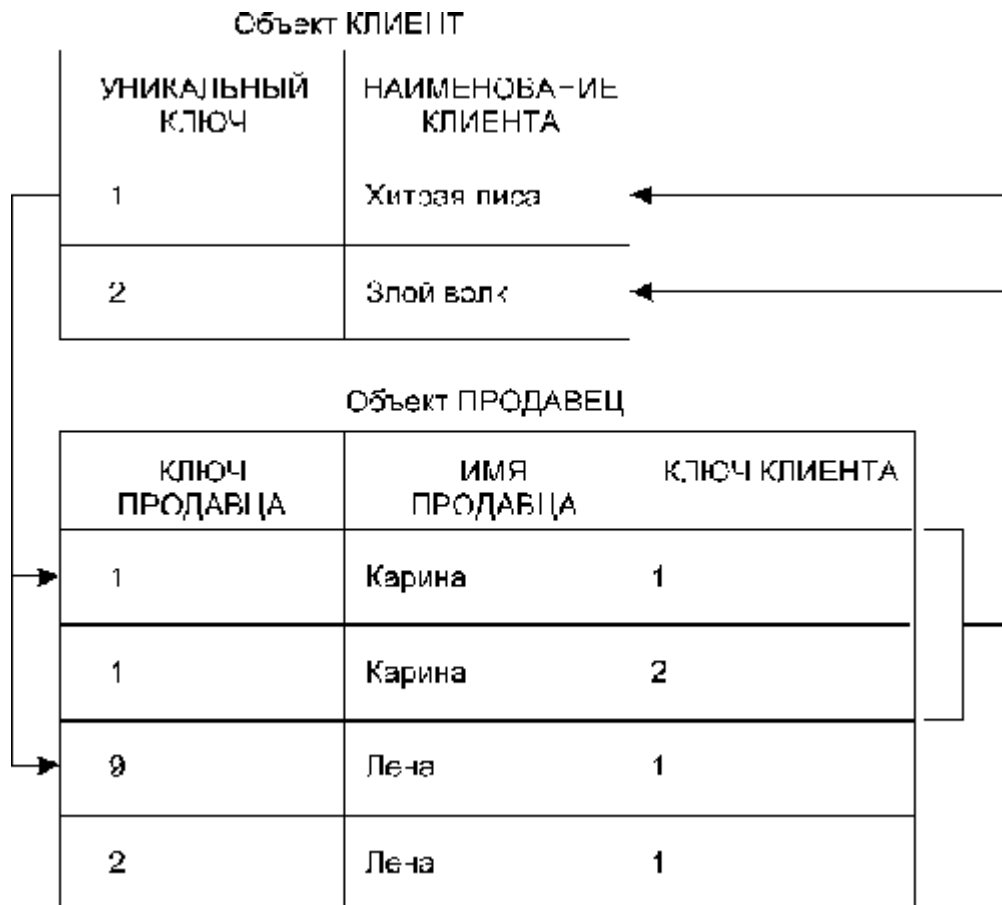


Рис. 2.5. Взаимосвязь между данными при отношении "многие ко многим"

Согласно теории реляционных баз данных для хранения взаимосвязи "многие ко многим" требуются три объекта: по одному для каждой сущности и один для хранения связей между ними (промежуточный объект). Промежуточный объект будет содержать идентификаторы связанных объектов, как это показано на рис. 2.6.



Рис. 2.6.

Взаимосвязи между объектами являются частью концептуальной модели и должны отображаться в базе данных.

Наряду с взаимосвязями между объектами существуют взаимосвязи между атрибутами объекта. Здесь также различают взаимосвязи типа "один к одному", "один ко многим" и "многие ко многим".

Взаимосвязь "один к одному" (между двумя атрибутами)

Мы предполагаем, что ключ (номер) клиента является его уникальным идентификатором, то есть он не изменяется и при последующих поступлениях заказов от данного клиента. Если наряду с номером клиента в базе данных хранится и другой его уникальный идентификатор (например, номер паспорта), то между такими двумя уникальными идентификаторами существует взаимосвязь "один к одному". На рис. 2.7,а эта взаимосвязь обозначается одинарными стрелками.

Взаимосвязь "один ко многим" (между двумя атрибутами)

Имя клиента и его номер существуют совместно. Клиентов с одинаковыми именами может быть много, но все они имеют различные номера. Каждому клиенту присваивается уникальный номер. Это означает, что данному номеру клиента соответствует только одно имя. Взаимосвязь "один ко многим" обозначается одинарной стрелкой в направлении к "одному" и двойной стрелкой в направлении ко "многим" (рис. 2.7,б).

Взаимосвязь "многие ко многим" (между двумя атрибутами)

Несколько клиентов с одинаковыми именами могли быть обслужены несколькими продавцами. Несколько продавцов с одинаковыми именами могли полу-чить заказы от нескольких клиентов.

Между атрибутами "имя клиента" и "имя продавца" существует взаимосвязь "многие ко многим". Мы обозначаем эту взаимосвязь двойными стрелками (рис. 2.7,в).

Типы моделей данных

Иерархическая и сетевая модели данных стали применяться в системах управления базами данных в начале 60-х годов. В начале 70-х годов была предложена реляционная модель данных. Эти три модели различаются в основном способами представления взаимосвязей между объектами.

Иерархическая модель данных строится по принципу иерархии типов объектов, то есть один тип объекта является главным, а остальные, находящиеся на низших уровнях иерархии, - подчиненными (рис. 2.8). Между главным и подчиненными объектами устанавливается взаимосвязь "один ко многим". Иными словами, для данного главного типа объекта существует несколько подчиненных типов объекта. В то же время для каждого экземпляра главного объекта может быть несколько экземпляров подчиненных типов объектов. Таким образом, взаимосвязи между объектами напоминают взаимосвязи в генеалогическом древе за единственным исключением: для каждого порожденного (подчиненного) типа объекта может быть только один исходный (главный) тип объекта.

На рис. 2.8 узлы и ветви образуют иерархическую древовидную структуру. Узел является совокупностью атрибутов, описывающих объект. Наивысший в иерархии узел называется корневым (это главный тип объекта). Корневой узел находится на первом уровне. Зависимые узлы (подчиненные типы объектов) находятся на втором, третьем и т. д. уровнях

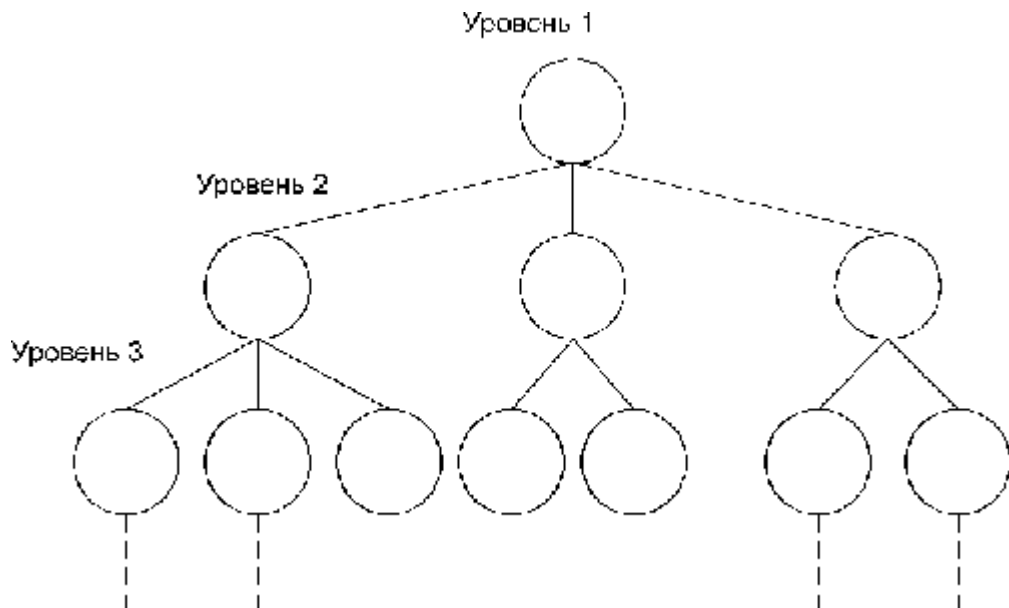


Рис. 2.8.

В сетевой модели данных понятия главного и подчиненных объектов несколько расширены. Любой объект может быть и главным и подчиненным (в сетевой модели главный объект обозначается термином "владелец набора", а подчиненный - термином "член набора"). Один и тот же объект может одновременно выступать и в роли владельца, и в роли члена набора. Это означает, что каждый объект может участвовать в любом числе взаимосвязей. Схема сетевой модели приведена на рис. 2.9. В реляционной модели данных объекты и взаимосвязи между ними представляются с помощью таблиц, как это показано на рис. 2.10. Взаимосвязи также рассматриваются в качестве объектов. Каждая таблица представляет один объект и состоит из строк и столбцов. В реляционной базе данных каждая таблица должна иметь первичный ключ (ключевой элемент) - поле или комбинацию полей, которые единственным образом идентифицируют каждую строку в таблице. Благодаря своей простоте и естественности представления реляционная модель получила наибольшее распространение в СУБД для персональных компьютеров. Все рассматриваемые в книге средства разработки пользовательских приложений поддерживают именно реляционную модель данных.

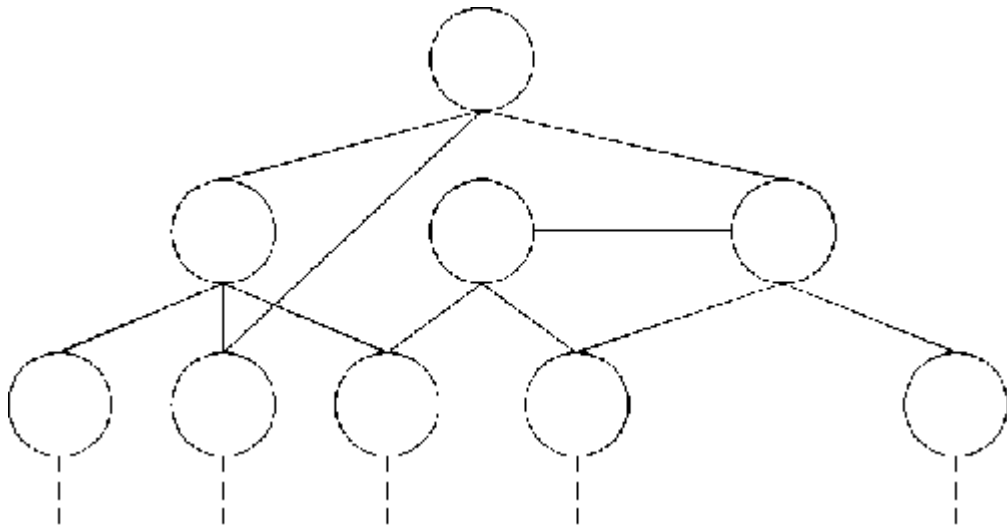


Рис. 2.9.

2.2. Проектирование базы данных

Все тонкости построения информационной модели преследуют одну-единственную цель - получить хорошую базу данных. А что это такое?

Существует очень простое понятие базы данных как большого по объему хранилища, в которое организация помещает все используемые ею данные и из которого различные пользователи могут их получать, используя различные приложения. Такая единая база данных представляется идеальным вариантом, хотя на практике это решение по различным причинам труднодостижимо. Поэтому чаще всего под базой данных понимают любой набор хранящихся в компьютере взаимосвязанных данных.

В этом параграфе мы изучим:

- методику проектирования базы данных на основе последовательного построения информационной модели;
- принципы нормализации данных;
- основные требования к реализации физической модели.

В основу проектирования БД должны быть положены представления конечных пользователей конкретной организации - концептуальные требования к системе. Именно конечный пользователь в своей работе принимает решения с учетом получаемой в результате доступа к базе данных информации. От оперативности и качества этой информации будет зависеть эффективность работы организации. Данные, помещаемые в базу данных, также предоставляет конечный пользователь.

При рассмотрении требований конечных пользователей необходимо принимать во внимание следующее:

- База данных должна удовлетворять актуальным информационным потребностям организации. Получаемая информация должна по структуре и содержанию соответствовать решаемым задачам.
- База данных должна обеспечивать получение требуемых данных за приемлемое время, то есть отвечать заданным требованиям производительности.
- База данных должна удовлетворять выявленным и вновь возникающим требованиям конечных пользователей.
- База данных должна легко расширяться при реорганизации и расширении предметной области.
- База данных должна легко изменяться при изменении программной и аппаратной среды.
- Загруженные в базу данных корректные данные должны оставаться корректными.
- Данные до включения в базу данных должны проверяться на достоверность.
- Доступ к данным, размещаемым в базе данных, должны иметь только лица с соответствующими полномочиями.

Этапы проектирования базы данных с учетом рассмотренных выше аспектов представлены на рис. 2.11.

В результате анализа поставленной заказчиком задачи и обработки требований конечных пользователей составляется концептуальная модель.

При разработке логической модели базы данных прежде всего необходимо решить, какая модель данных наиболее подходит для отображения конкретной концептуальной модели предметной области. Коммерческие системы управления базами данных поддерживают одну из известных моделей данных или некоторую их комбинацию. Почти что все популярные системы для персональных компьютеров поддерживают реляционную модель данных.

Отображение концептуальной модели данных на реляционную модель производится относительно просто. Каждый прямоугольник концептуальной модели отображается в одно отношение, которое отражает представление пользователя в удобном для него табличном формате. Простота отображения связана с тем, что при разработке концептуальной модели использовался реляционный подход.

Рассмотрим этапы проектирования базы данных, которые должны обеспечить необходимую независимость данных и выполнение эксплуатационных требований (пожеланий пользователей).

Этап 1. Определение сущностей

Исходя из задачи, описанной в [первой главе](#), выделим следующие сущности:

1. МОДЕЛЬ;
2. АВТОМОБИЛЬ;
3. КЛИЕНТ;
4. ПРОДАВЕЦ;
5. ЗАКАЗ;
6. ПРОДАЖА;
7. СЧЕТ.

Этап 2. Определение взаимосвязей между сущностями

Определим для включенных в модель сущностей взаимосвязи в соответствии с рекомендациями, данными в предыдущем параграфе. Полученная после этого информационная модель представлена на рис. 2.12.

Необходимо отметить что на рис. 2.2,а взаимосвязь между объектами КЛИЕНТ и ЗАКАЗ рассматривается в определенный момент времени, для примера связи "один к одному". Однако анализируя данную взаимосвязь более широко, получим, что один клиент в разное время может производить несколько заказов. С другой стороны, один заказ принадлежит только одному клиенту и поэтому на рис. 2.12 между сущностями КЛИЕНТ и ЗАКАЗ установлена взаимосвязь "один ко многим".

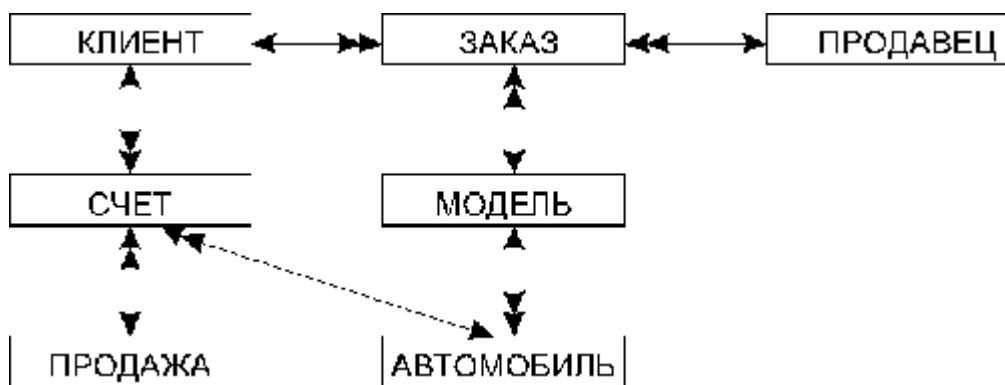


Рис. 2.12.

Этап 3. Задание первичных и альтернативных ключей, определение атрибутов сущностей

Для каждой сущности определим атрибуты, которые мы будем хранить в БД. При этом необходимо учитывать тот факт, что при переходе от логической к физической модели данных может произойти усечение числа объектов. На самом деле, как правило, значительное число данных, необходимых пользователю, может быть достаточно легко подсчитано в момент вывода информации. В то же время, в связи с изменением алгоритмов расчета или исходных величин, некоторые расчетные показатели приходится записывать в БД, чтобы гарантированно обеспечить

фиксацию их значений. Выбор показателей, которые обязательно следует хранить в БД, достаточно сложен. Нечасто можно найти однозначное решение этой проблемы, и в любом случае оно потребует тщательного изучения работы предприятия и анализа концептуальной модели.

Атрибуты, включаемые в состав БД для рассматриваемой модели, приведены в табл. 2.1. Информационная модель после третьего этапа проектирования приведена на рис. 2.13.

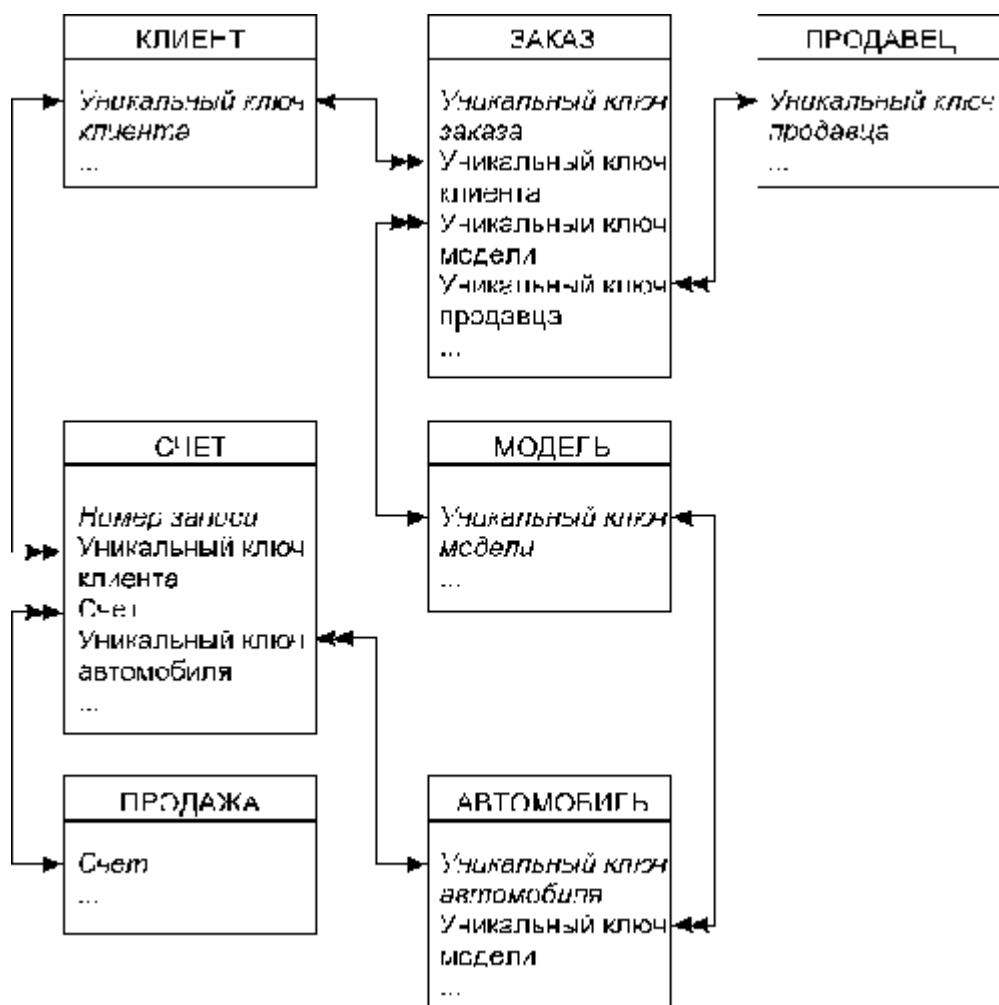


Рис. 2.13.

Таблица 2.1. Атрибуты и первичные ключи сущностей информационной модели

Сущность	Первичный ключ	Атрибуты
МОДЕЛЬ	Уникальный ключ модели	Уникальный ключ модели Наименование модели Наименование фирмы Наименование страны Рабочий объем двигателя Количество цилиндров Мощность Крутящий момент Наименование топлива Максимальная скорость

		Время разгона до 100 км/ч
		Наименование шин
		Наименование кузова
		Количество дверей
		Количество мест
		Длина
		Ширина
		Высота
		Расход топлива при 90 км/ч
		Расход топлива при 120 км/ч
		Расход топлива при городском цикле
АВТОМОБИЛЬ	Уникальный ключ автомобиля	Уникальный ключ автомобиля
		Уникальный ключ модели
		Дата выпуска
		Стоимость
КЛИЕНТ	Уникальный ключ клиента	Уникальный ключ клиента
		Наименование клиента
		Адрес
		Телефон
		Факс
		Фамилия
		Имя
		Отчество
		Признак юридического лица
		Примечание
ПРОДАЖА	Счет	Счет
		Дата продажи
		Сумма
СЧЕТ	Номер записи	Номер записи
		Счет
		Уникальный ключ клиента
		Уникальный ключ автомобиля
		Дата выписки
		Пометка об оплате
		Сумма
ЗАКАЗ	Уникальный ключ заказа	Уникальный ключ заказа
		Уникальный ключ клиента
		Уникальный ключ модели
		Уникальный ключ продавца

ПРОДАВЕЦ	Уникальный ключ	Уникальный ключ
	продавца	продавца
		Фамилия
		Имя
		Отчество

Этап 4. Приведение модели к требуемому уровню нормальной формы

Приведение модели к требуемому уровню нормальной формы является основой построения реляционной БД.

В процессе нормализации элементы данных группируются в таблицы, представляющие объекты и их взаимосвязи. Теория нормализации основана на том, что определенный набор таблиц обладает лучшими свойствами при включении, модификации и удалении данных, чем все остальные наборы таблиц, с помощью которых могут быть представлены те же данные. Введение нормализации отношений при разработке информационной модели обеспечивает минимальный объем физической, то есть записанной на каком-либо носителе, БД, и ее максимальное быстроедействие, что напрямую отражается на качестве функционирования информационной системы. Нормализация информационной модели выполняется в несколько этапов.

Данные, представленные в виде двумерной таблицы, являются первой нормальной формой реляционной модели данных.

Первый этап нормализации заключается в образовании двумерной таблицы, содержащей все необходимые атрибуты информационной модели, и в выделении ключевых атрибутов. Очевидно, что полученная весьма внушительная таблица будет содержать очень разнородную информацию. В этом случае будут наблюдаться аномалии включения, обновления и удаления данных, так как при выполнении этих действий нам придется уделить внимание данным (вводить или заботиться о том, чтобы они не были стерты), которые не имеют к текущим действиям никакого отношения. Например, может наблюдаться такая парадоксальная ситуация. При включении в каталог продаж новой модели автомобиля нам сразу придется указать купившего ее клиента.

Отношение задано во второй нормальной форме, если оно является отношением в первой нормальной форме и каждый атрибут, не являющийся первичным атрибутом в этом отношении, полностью зависит от любого возможного ключа этого отношения.

Если все возможные ключи отношения содержат по одному атрибуту, то это отношение задано во второй нормальной форме, так как в этом случае все атрибуты, не являющиеся первичными, полностью зависят от возможных ключей. Если ключи состоят более чем из одного атрибута, отношение, заданное в первой нормальной форме, может не быть отношением во второй нормальной форме. Приведение отношений ко второй нормальной форме заключается в обеспечении полной функциональной зависимости всех атрибутов от ключа за счет разбиения таблицы на несколько, в которых все имеющиеся атрибуты будут иметь полную функциональную зависимость от ключа этой таблицы. В процессе приведения модели ко второй нормальной форме в основном исключаются аномалии дублирования данных.

Отношение задано в третьей нормальной форме, если оно задано во второй нормальной форме и каждый атрибут этого отношения, не являющийся первичным, не транзитивно зависит от каждого возможного ключа этого отношения.

Транзитивная зависимость выявляет дублирование данных в одном отношении. Если А, В и С - три атрибута одного отношения и С зависит от В, а В от А, то говорят, что С транзитивно зависит от А, как это схематично показано на рис. 2.14,а. Преобразование в третью нормальную форму происходит за счет разделения исходного отношения на два, как это показано на рис. 2.14,б.

Например, если все данные о моделях автомобилей и самих поступающих автомобилях хранятся в одном отношении, то для нескольких автомобилей одной модели пришлось бы

многократно указывать тип кузова, количество дверей и другие технические характеристики. В этом случае технические характеристики зависят от модели автомобиля и при наличии нескольких автомобилей одной модели будут дублироваться. Дублирование исчезает, если из одного отношения выделить отношение, в котором будут храниться данные о моделях, и отношение, в котором будут храниться данные об автомобилях.

Существуют и более высокие формы нормализации, но авторы не встречались с необходимостью их применения за достаточно длительную историю создания систем обработки данных и поэтому сочли возможным уберечь читателя от потока теории.

Давайте сформулируем основные правила, которым нужно следовать при проектировании базы данных:

Исключайте повторяющиеся группы - для каждого набора связанных атрибутов создайте отдельную таблицу и снабдите ее первичным ключом. Выполнение этого правила автоматически приведет ко второй нормальной форме. Помимо теоретических указаний в этом правиле есть и чисто практический смысл. Представьте, что в вашем списке заказов вы указываете имена ваших клиентов. Клиент "Хитрая лиса" достаточно активен и часто делает у вас заказы. Бьемся об заклад, что найдутся считанные люди, которые в десяти упоминаниях напишут это имя абсолютно одинаково. Ну кто-то где-нибудь да напишет "Хитрый лис", а для СУБД это уже другой клиент. Поэтому гораздо лучше вести список своих клиентов в отдельной таблице, а в списке заказов использовать только присвоенные им уникальные идентификаторы.

Исключайте избыточные данные - если атрибут зависит только от части составного ключа, переместите атрибут в отдельную таблицу. Это правило помогает избежать потери одних данных при удалении каких-то других. Везде, где возможно использование идентификаторов вместо описания, выносите в отдельную таблицу список идентификаторов с пояснениями к ним.

Исключайте столбцы, которые не зависят от ключа - если атрибуты не вносят свою лепту в описание ключа, переместите их в отдельную таблицу.

С учетом выше изложенного в нашей модели необходимо видоизменить список атрибутов сущности МОДЕЛЬ и добавить такие новые сущности, как ТОПЛИВО, ШИНЫ, КУЗОВ, ФИРМА, СТРАНА (рис. 2.15).

В основном изменения в модели связаны с введением искусственных атрибутов, которые в виде кодов участвуют в отношениях вместо естественных атрибутов (вид топлива, марка шин и т. п.). К необходимости введения в модель искусственных атрибутов мы пришли в процессе нормализации. В общем случае мы рекомендуем использовать вместо естественных атрибутов коды в следующих случаях:

В предметной области может наблюдаться синонимия, то есть естественный атрибут отношения не обладает свойством уникальности. Например, среди сотрудников фирмы могут быть однофамильцы или даже полные тезки. В этом случае решить проблему помогает уникальный табельный номер.

Если отношение участвует во многих связях, то для их отображения создается несколько таблиц, в каждой из которых повторяется идентификатор отношения. Для того чтобы не использовать во всех таблицах длинный естественный атрибут объекта, можно применять более короткий код. Это также будет способствовать повышению быстродействия вашей системы.

Если естественный атрибут может изменяться во времени (например, фамилия), то это может вызвать очень большие сложности при эксплуатации системы. Представьте, что ваш лучший продавец, очаровательная девушка Карина, вышла замуж. Что будет с данными, которые привязаны к ее девичьей фамилии? Использование неизменяемого кода (табельного номера) позволит избежать этих сложностей.

Атрибуты, включаемые в измененные или добавленные в модель сущности, приведены в табл. 2.2.

Таблица 2.2. Атрибуты и первичные ключи измененных или добавленных сущностей информационной модели

Сущность	Первичный ключ	Атрибуты
МОДЕЛЬ	Уникальный ключ модели	Уникальный ключ модели Наименование модели Уникальный ключ фирмы Наименование страны Рабочий объем двигателя Количество цилиндров Мощность

		Крутящий момент
		Уникальный ключ топлива
		Максимальная скорость
		Время разгона до 100 км/ч
		Уникальный ключ шин
		Уникальный ключ кузова
		Количество дверей
		Количество мест
		Длина
		Ширина
		Высота
		Расход топлива при 90 км/ч
		Расход топлива при 120 км/ч
		Расход топлива при городском цикле
ТОПЛИВО	Уникальный ключ топлива	Уникальный ключ топлива Наименование топлива
ШИНЫ	Уникальный ключ шин	Уникальный ключ шин Наименование шин
КУЗОВ	Уникальный ключ кузова	Уникальный ключ кузова Наименование кузова
ФИРМА	Уникальный ключ фирмы	Уникальный ключ фирмы Наименование фирмы
СТРАНА	Уникальный ключ страны	Уникальный ключ страны Наименование страны

Этап 5. Физическое описание модели

На этом этапе мы должны составить проекты таблиц, которые будут в дальнейшем реализовываться в конкретной СУБД. Назначения имен таблиц и их атрибутов отражены в табл. 2.3.

Таблица 2.3. Проект таблиц для физической модели
Model (Модель) + **1**

№ п/п	Наименование поля	Примечание
1	Key_model	Уникальный ключ модели
2	Name_model	Наименование модели
3	Key_firm	Уникальный ключ фирмы
4	Swept_volume	Рабочий объем, см ³

5	Quantity_drum	Количество цилиндров
6	Capacity	Мощность, л. с.
7	Torgue	Крутящий момент
8	Key_fuel_oil	Уникальный ключ топлива
9	Top_speed	Максимальная скорость,, км/ч
10	Starting	Время разгона до 100 км/ч,, с
11	Key_tyre	Уникальный ключ шин
12	Key_body	Уникальный ключ кузова
13	Quantity_door	Количество дверей
14	Quantity_sead	Количество мест
15	Length	Длина, мм
16	Width	Ширина, мм
17	Height	Высота, мм
18	Expense_90	Расход топлива при 90 км/ч, л/100 км
19	Expense_120	Расход топлива при 120 км/ч, л/100 км
20	Expense_town	Расход топлива при городском цикле, л/100 км

Firm (Фирма)		
№ п/п	Наименование поля	Примечание
1	Key_firm	Уникальный ключ фирмы
2	Name_firm	Наименование фирмы
3	Key_country	Уникальный ключ страны

Country (Страна)		
№ п/п	Наименование поля	Примечание
1	Key_country	Уникальный ключ страны
2	Name_country	Наименование страны

Fuel_oil (Топливо)		
№ п/п	Наименование поля	Примечание
1	Key_fuel_oil	Уникальный ключ топлива
2	Name_fuel_oil	Наименование топлива

Tyre (Шины)		
№ п/п	Наименование поля	Примечание
1	Key_tyre	Уникальный ключ шин
2	Name_tyre	Наименование шин

Body (Кузов)		
№ п/п	Наименование поля	Примечание
1	Key_body	Уникальный ключ кузова
2	Name_body	Наименование кузова

Automobile passenger car (Автомобиль)		
№ п/п	Наименование поля	Примечание
1	Key_auto	Уникальный ключ автомобиля
2	Key_model	Уникальный ключ модели
3	Date_issue	Дата выпуска в формате ДД.ММ.ГГ
4	Cost	Стоимость,, \$

Customer (Клиент)		
№ п/п	Наименование поля	Примечание
1	Key_custome	Уникальный ключ клиента
2	Name_customer	Наименование клиента
3	Address	Адрес
4	Tel	Телефон
5	Fax	Факс
6	Last_name	Фамилия
7	First_name	Имя
8	Patronymic	Отчество
9	Juridical	Признак юридического лица
10	Comment	Примечание

Sale (Продажа)		
№ п/п	Наименование поля	Примечание
1	Account	Счет
2	Date_sale	Дата продажи в формате

		ДД.ММ.ГГ
3	Sum_	Сумма, \$
Account (Счет)		
	+	10
№ п/п	Наименование поля	Примечание
1	Number_record	Номер записи
2	Account_	Счет
3	Key_customer	Уникальный ключ клиента
4	Key_auto	Уникальный ключ автомобиля
5	Date_write	Дата выписки в формате ДД.ММ.ГГ
6	Selled	Пометка об оплате
7	Sum_	Сумма, \$
Order_ (Заказ)		
	+	11
№ п/п	Наименование поля	Примечание
1	Key_order	Уникальный ключ заказа
2	Key_customer	Уникальный ключ клиента
3	Key_model	Уникальный ключ модели
4	Key_salman	Уникальный ключ продавца
Salesman (Продавец)		
	+	12
№ п/п	Наименование поля	Примечание
1	Key_salman	Уникальный ключ продавца
2	Last_name	Фамилия
3	First_name	Имя
4	Patronymic	Отчество

На этапе физического проектирования мы должны задуматься о такой серьезной проблеме, как обеспечение безошибочности и точности информации, хранящейся в БД. Это называется обеспечением целостности базы данных.

Обеспечением целостности базы данных называется система мер, направленных на поддержание правильности данных в базе в любой момент времени.

Мы привыкли доверять достоверности данных, помещаемых в печатных изданиях. При подготовке книги помещаемые в нее данные проверяют несколько редакторов. Они же стараются сделать так, чтобы книга была написана литературным языком и соответствовала неким нормам, с которыми читатели подходят к книгам различного жанра. Удивительно было бы при чтении детектива встретить ссылки на научные источники в конце страницы.

При общении человека с компьютером наблюдается некий стойкий феномен. Мы приравниваем компьютер к книге из издательства с самой известной маркой. Мы априори верим всем данным, которые выдает нам это славное устройство. Мы не хотим ничего слышать о программных

ошибках, сбоях и неправильном вводе данных. Вы встречали человека, размахивающего листком с распечаткой и неустанно повторяющего как молитву фразу: "Это я рассчитал на компьютере!" Мы встречали, и не раз.

Если же трезво взглянуть на проблему, ситуация начинает выглядеть совершенно иначе. При подготовке книги затраты на ее редактирование составляют незначительную часть от общих затрат на ее выпуск в свет. В системах обработки данных наоборот. Затраты на проверку и поддержание достоверности данных могут составлять значительную часть от общих эксплуатационных затрат. Например, в транспортных предприятиях для контроля правильности ввода данных с путевой документации практикуется параллельный ввод одних и тех же данных несколькими операторами. Считается, что вероятность совершения одной и той же ошибки в этом случае будет крайне мала и простое сравнение результатов ввода различных операторов поможет получить безошибочные данные.

Естественно появляется соблазн избежать дорогостоящей верификации данных. Как сказал один рабочий судоверфи другому, объясняя, почему он не заделал дыру в корпусе корабля: "Ее никто не увидит, она же под водой".

Ошибки в данных неприятны тем, что они остаются незамеченными до тех пор, пока не приведут к тяжелым последствиям, если только вы не позаботитесь обнаружить эти ошибки раньше. Достаточно убедительный довод, чтобы заранее воспользоваться предоставляемыми СУБД мерами для блокирования появления возможных ошибок в разрабатываемой базе данных.

В СУБД целостность данных обеспечивается набором специальных предложений, называемых ограничениями целостности.

Ограничения целостности - это набор определенных правил, которые устанавливают допустимость данных и связей между ними.

Ограничения целостности в большинстве случаев определяются особенностями предметной области. Например, мощность двигателя серийного легкового автомобиля вряд ли может быть ниже 30 л. с.

Ограничения целостности могут относиться к разным объектам БД: атрибутам (полям), записям, отношениям, связям между ними и т. п. Для полей могут использоваться следующие виды ограничений:

- Тип и формат поля автоматически допускают ввод только данных определенного типа. Выбор типа поля Date в формате ДД.ММ.ГГ позволит пользователю ввести только шесть чисел. При этом первая пара цифр не сможет превысить в лучшем случае значения 31, а вторая - 12.
- Задание диапазона значений, как правило, используется для числовых полей. Диапазон допустимых значений может быть ограничен с двух сторон (закрытый диапазон), а может с какой-то одной: верхней или нижней (открытый диапазон).
- Недопустимость пустого поля позволяет избежать появления в БД "ничейных" записей, в которых пропущены какие-либо обязательные атрибуты.
- Задание списка значений позволяет избежать излишнего разнообразия данных, если его можно ограничить. Например, для указания типа кузова мы можем ограничить фантазию пользователя только общепринятыми названиями: Седан, кабриолет и т. д.
- Проверка на уникальность значения какого-то поля позволяет избежать записей-дубликатов. Вряд ли будет удобно в справочнике клиентов иметь несколько записей для одного и того же лица.

Для реализации ограничений целостности, имеющих отношение к записи, таблицам или связям между ними, в СУБД используются триггеры.

Как конкретно реализуется в каждой из рассматриваемых СУБД ограничение целостности мы увидим на примере создания БД в [шестой главе](#).

2.3. Словарь данных

Управленческим инструментарием разработки при проектировании базы данных является словарь данных (СД).

В этом параграфе мы познакомимся с тем, как правильно использовать возможности словаря данных при проектировании и эксплуатации БД.

Внедрение базы данных на любом предприятии занимает довольно продолжительное время. Ее расширение происходит по мере разработки и интеграции используемых прикладных

программ. В процессе эксплуатации вводятся новые элементы данных, а те, которые использовались при проектировании базы данных, могут быть изменены.

Преимущества использования СД заключаются в эффективном накоплении, определении и управлении суммарным ресурсом данных предметной области. Словарь данных призван помогать пользователю в выполнении следующих функций:

- совместное использование данных с другими пользователями;
- осуществление простого и эффективного управления элементами данных при вводе в систему новых элементов или изменении описания существующих;
- уменьшение избыточности и противоречивости данных;
- определение степени влияния изменений в элементах данных на всю базу данных;
- централизация управления элементами данных с целью упрощения проектирования базы данных и ее расширения.

Идеальный СД содержит также сведения и о других категориях данных, таких, как группы элементов данных, корреспондирующие базы данных и перекрестные ссылки между группами элементов данных и базами данных. Кроме того, в нем фиксируются сведения о тех прикладных программах, которые используют базу данных, и имеются сведения о кодах защиты информации и разграничении доступа.

Прежде всего остановимся на вопросах использования СД при проектировании системы обработки данных.

На первом этапе проектирования базы данных необходимо собрать сведения о предметной области, в том числе о назначении, способах использования и о структуре данных, а по мере развития проекта осуществлять централизованное накопление информации о концептуальной, логической, внутренней и внешних моделях данных. Словарь данных является как раз тем средством, которое позволяет при проектировании, эксплуатации и развитии базы данных поддерживать и контролировать информацию о данных.

При сборе информации о данных следует установить правила присвоения имен элементам, добиться однозначного толкования различными подразделениями назначения источников и соглашений по присвоению имен, сформулировать приемлемые для всех пользователей описания элементов данных и выявить синонимы. Этот процесс включает несколько итераций и связан с необходимостью разрешения конфликтных ситуаций. Отдельные подразделения подчас переоценивают свою роль на предприятии, что приводит при разработке информационной системы к конфликтам. Разработчику в таких случаях придется выступать в роли арбитра. Если вам не по душе слушать крики: "Судью на мыло!", то для обеспечения эффективного сбора и накопления информации о данных желательно, чтобы все, кто имеет отношение к базе данных, пользовались автоматизированным словарем данных.

Словарь данных содержит информацию об источниках, форматах и взаимосвязях между данными, их описания, сведения о характере использования и распределении ответственности. Словарь данных можно рассматривать как "метабазу данных", в которой хранится информация о базе данных.

Одно из главных назначений словаря данных состоит в документировании данных. Так как база данных обслуживает множество пользователей, крайне необходимо, чтобы они правильно понимали, что представляют собой данные.

Проектировщик базы данных рассматривает различные характеристики данных. На ранней стадии проектирования прежде всего готовятся описания элементов данных на естественном языке. Эти описания или определения должны быть точными, недвусмысленными и согласованными.

Например, если три подразделения используют одни и те же данные по-разному, совсем не просто сформулировать приемлемое для всех одно описание разделяемого элемента. Поиск решений проблем такого рода является прерогативой администратора БД. Это часто создает конфликтные ситуации, разрешить которые гораздо сложнее, чем технические вопросы применения базы данных.

На этой стадии разработки текстовых описаний данных проектировщик абстрагируется от способа их физического представления в базе данных. В частности, ему не следует определять, как хранить данные - в упакованном, символьном или каком-либо другом виде.

Накопление информации в словаре данных целесообразно начинать уже на самой ранней стадии проектирования. В процессе работы разработчики выясняют у пользователей, какой должна быть система, какие данные будут входными, какого рода информацию они хотят получить из системы, вводя имена элементов данных, например "номер счета", "остаток" или "процент" в банковской системе. При этом обе стороны должны трактовать используемые термины однозначно, иначе может случиться так, что разработанная система не будет удовлетворять требованиям пользователей. Поэтому второе важное назначение словаря данных - обеспечить эффективное взаимодействие между различными категориями разработчиков и пользователей.

Рассмотрим следующий пример. В банковской системе одним из центральных элементов данных является "остаток". Каков остаток на данном счете? Для большинства неспециалистов ответ очевиден. Однако в главной таблице счетов соответствующей системы в записи по одному счету может храниться до двадцати пяти полей, в именах которых присутствует термин "остаток". Поэтому важно, чтобы и пользователь и разработчик представляли себе, какой именно остаток имеется в виду: "остаток на счете на начало дня", "остаток на счете на конец вчерашнего дня", "фактический остаток" или "остаток на сберегательной книжке". "Остаток на сберегательной книжке" увеличивается сразу после того, как клиент делает вклад, но если вклад сделан с помощью чека (вдруг вы разрабатываете систему обработки данных для использования на ужасном Западе), то "фактический остаток" увеличится только после оплаты чека. На самом деле существует гораздо больше различных полей "остаток". Словарь данных может использоваться для централизованного накопления информации обо всех элементах данных и для обеспечения эффективного взаимодействия между всеми участниками проекта.

Таким образом, два важнейших назначения словаря данных состоят в централизованном ведении и управлении данными как ресурсом на всех этапах проектирования, реализации и эксплуатации системы, а также в обеспечении эффективного взаимодействия между всеми участниками проекта.

В случае распределенной базы данных вся она или ее отдельные части могут размещаться на удаленных друг от друга вычислительных машинах, соединенных линиями связи. Одни рабочие станции в сети могут обращаться только к локальной базе данных, а другие - как к локальной, так и к внешним.

В этом случае в словарь данных может быть введена информация обо всех местах физического хранения данных, а также ограничения секретности, безопасности и доступа. С помощью этой информации словарь данных может "решить", каким образом удовлетворить запрос пользователя: обратиться к локальной базе данных или, если пользователь обладает соответствующими полномочиями, передать запрос на внешнюю ПЭВМ.

В настоящей книге словари данных рассматриваются в контексте совместного использования с СУБД. Существует мнение, что словарь данных можно вести "вручную на бумаге". Однако неавтоматизированный словарь данных не может обеспечить получение по-разному отсортированных списков элементов данных, которыми пользуются разработчики. Один и тот же элемент может неодинаково использоваться в различных приложениях. На ранней стадии проектирования выявляются далеко не все связи между данными. Впоследствии обнаруживается, что данные применяются в разнообразных приложениях. Они могут встречаться, например, во входных и выходных форматах, связанных между собой, и всякий раз рассматриваются в различных контекстах. Чтобы учесть все возможные ограничения, необходимо приложить значительные усилия. Процесс проектирования же становится в таком случае трудно управляемым. Гораздо проще организовать и управлять разработкой с помощью автоматизированного словаря данных.

Для успешного применения словаря данных при разработке системы следует централизовать накопление информации в этом - едином - источнике, из которого программисты смогут копировать описания структур данных и включать их в свои программы на всех этапах проектирования. В случае применения "ручного" или не интегрированного словаря в нем время от времени может происходить нарушение непротиворечивости информации по отношению к фактическому состоянию системы.

В идеальном случае интерфейс между СУБД и словарем данных должен обеспечивать доступ системы словаря к справочникам СУБД, в которых хранится информация о ее текущем состоянии. Модификация типов данных может производиться только после того, как это будет зарегистрировано в словаре данных. Обновление самих данных допускается лишь после проверки их корректности средствами СУБД. Таким образом, словарь данных, СУБД и база данных образуют замкнутый контур.

В идеале словарь данных должен быть неотъемлемой составной частью всей системы обработки данных. За ввод данных в словарь ответственность несет администратор БД. Поскольку словарь данных является центральным звеном системы, необходимо постоянно поддерживать его копию, которая может использоваться для восстановления словаря после возникновения отказа всей системы или в случае непреднамеренного разрушения его рабочей версии. За сохранность словаря данных как жизненно важной части системы с базой данных полностью отвечает администрация базы данных.

Если словарь данных применяется для разграничения доступа к базе данных, то доступ к нему надо также разграничить. Следует строго ограничить круг лиц, которым разрешено модифицировать словарь данных. В отношении хранимой в словаре информации должен быть реализован режим секретности.

Внедрение словаря данных должно быть хорошо продумано по времени и объему, так как неверный шаг здесь может привести к неудаче всего проекта системы обработки данных. В то же время не следует думать, что существуют готовые решения, которые подходят для всех без исключения предприятий. Как и другие планируемые действия в системе обработки данных, план реализации словаря зависит от особенностей предметной области.

Подчас не только реализация всеобъемлющего словаря данных, но и создание интегрированной базы данных всей предметной области может оказаться преждевременным. Ведение словаря данных предполагает обучение пользователей, и до тех пор, пока в этом направлении не будут достигнуты определенные успехи, его установку производить не следует. В качестве первого приложения словаря данных можно избрать один из следующих вариантов.

Традиционная прикладная система. Словарь данных может применяться в прикладной системе средних размеров, не слишком тесно взаимодействующей с другими системами. Кроме того, система должна решать важные для предприятия задачи. И наконец, она должна быть развивающейся. Только при этих условиях могут быть использованы все преимущества использования словаря данных.

Несколько приложений для обработки данных. Скорее всего эксплуатируемые с использованием СУБД прикладные программы имеют большое значение для предприятия и, кроме того, со временем претерпевают определенные изменения. Если предполагается расширить круг прикладных программ, то внедрение системы словаря данных позволит использовать его возможности по обеспечению взаимодействия между пользователями, в области централизованного накопления информации, а также при подготовке и распространении документации.

Новые прикладные программы. Если предприятие планирует расширить круг используемых прикладных программ, то это самый подходящий случай для внедрения словаря данных. Программисты не всегда хорошо относятся к необходимости документирования уже существующих прикладных программ. Однако при разработке новых прикладных систем эта работа выполняется впервые, и документирование новых приложений неизбежно.

На прилагаемой к книге дискете вы найдете пример словаря данных для рассматриваемой в качестве примера системы обработки данных AUTO STORE.

С помощью этой программы вы сможете создать базу данных и словарь данных, где будет отражаться структура БД Auto_Store. В этой же программе после построения СД вы сможете проектировать словарь данных, а именно: редактировать имена полей таблиц, изменять их тип, разрядность, правила проверки, текст сообщения, значение по умолчанию, заголовок поля, комментарий для поля, устанавливать значение NULL, редактировать имена индексов, изменять их тип, трансформировать имена таблиц и наглядно анализировать связи между таблицами.

Данная программа позволяет создавать ту же БД Auto_Store на MS SQL Server, а также регистрировать пользователей на MS SQL Server и предоставлять определенные права доступа к БД Auto_Store или, точнее, права доступа к полям таблиц базы данных. Права доступа каждого из пользователей можно просмотреть, выбрав пункт "Права" из главного пункта меню "Разработка и проектирование".

Вход в программу. После запуска программы на экране появится форма, показанная на рис. 2.16. Для дальнейшей работы с программой необходимо ввести пароль. Пароль соответствует имени пользователя, что можно узнать из подсказки. При успешном наборе пароля откроется доступ к кнопке "Вход". Выбор пользователей производится из списка, находящегося на панели инструментов (рис. 2.17). Обратите внимание на уровень доступа, их всего пять.



Рис. 2.16. Вход в программу



Рис. 2.17. Панель инструментов для выбора пользователя

Создание базы данных. Здесь все очень просто. Выбираем пункт "Создание Базы Данных" из главного пункта меню "Создание и построение" и в открывшейся форме щелкаем по кнопке "Старт". На рис. 2.18 изображена форма в момент создания базы данных. Не забудьте щелкнуть по кнопке "Финиш" по завершении создания базы. После выполнения описанной процедуры предоставится возможность построения словаря данных.

Построение словаря данных. Аналогично строится словарь данных. Выбираем пункт "Построение Словаря Данных" из главного пункта меню "Создание и построение" и в открывшейся форме щелкаем по кнопке "Старт". На рис. 2.19 изображена форма в момент построения словаря данных. После щелчка по кнопке "Финиш" можно начинать проектирование словаря данных.

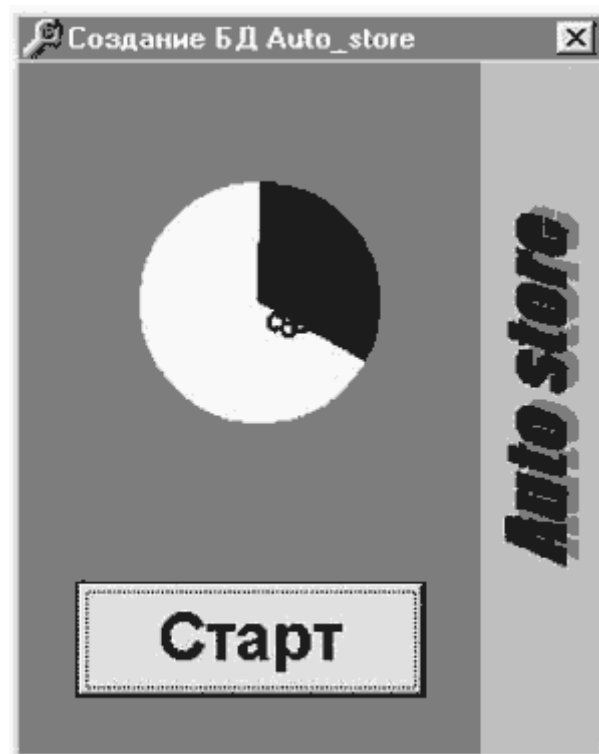


Рис. 2.18.



Рис. 2.19

Словарь данных. На рис. 2.20 показана форма словаря данных, предусматривающая следующие действия:

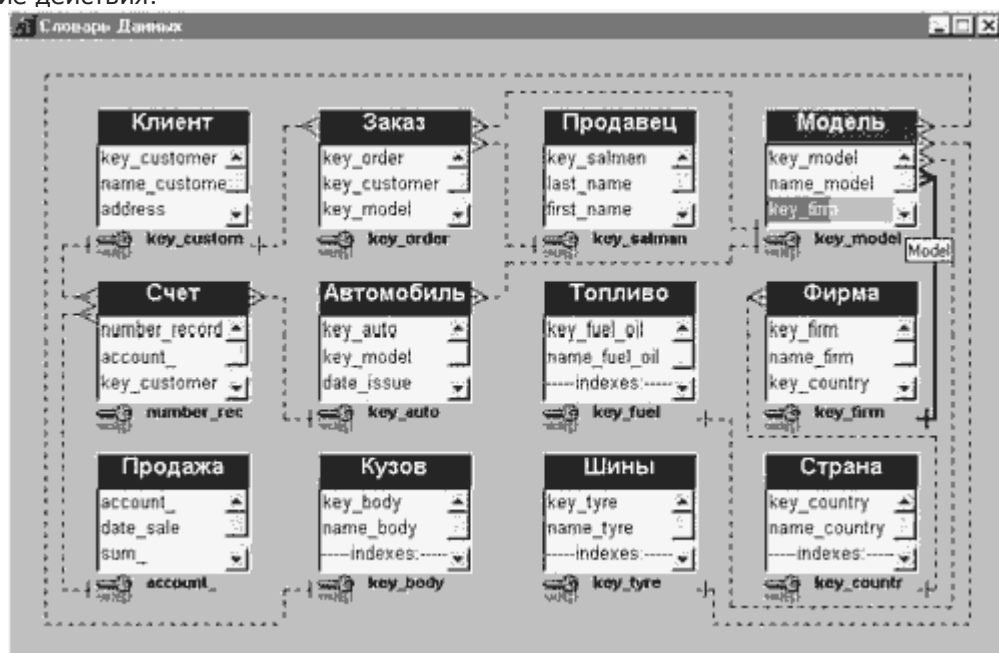


Рис. 2.20.

- Щелчком левой кнопки мыши по заголовку таблицы активизируется список полей данной таблицы.
- Щелчок правой кнопки мыши по заголовку таблицы приводит к замене наименования заголовка на физическое наименование таблицы, если перед этим заголовок имел вид русского эквивалента наименования таблицы, и, наоборот, изменяет наименование заголовка русским эквивалентом наименования таблицы, если заголовок имел вид наименования таблицы.
- Двойной щелчок левой кнопки мыши по заголовку таблицы позволяет переименовать заголовок таблицы (рис. 2.21).

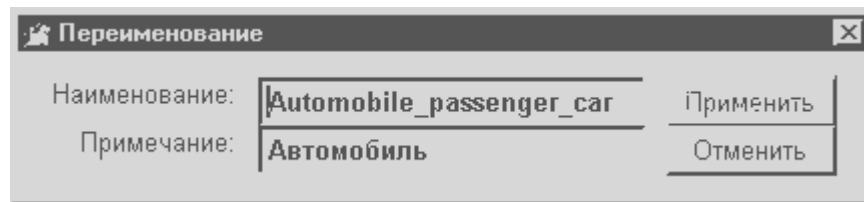


Рис. 2.21.

- Щелчок левой кнопки мыши по списку полей таблицы активизирует список полей данной таблицы.
- Двойной щелчок левой кнопки мыши по списку полей таблицы вызывает Конструктор таблицы на вкладке поля (рис. 2.22).

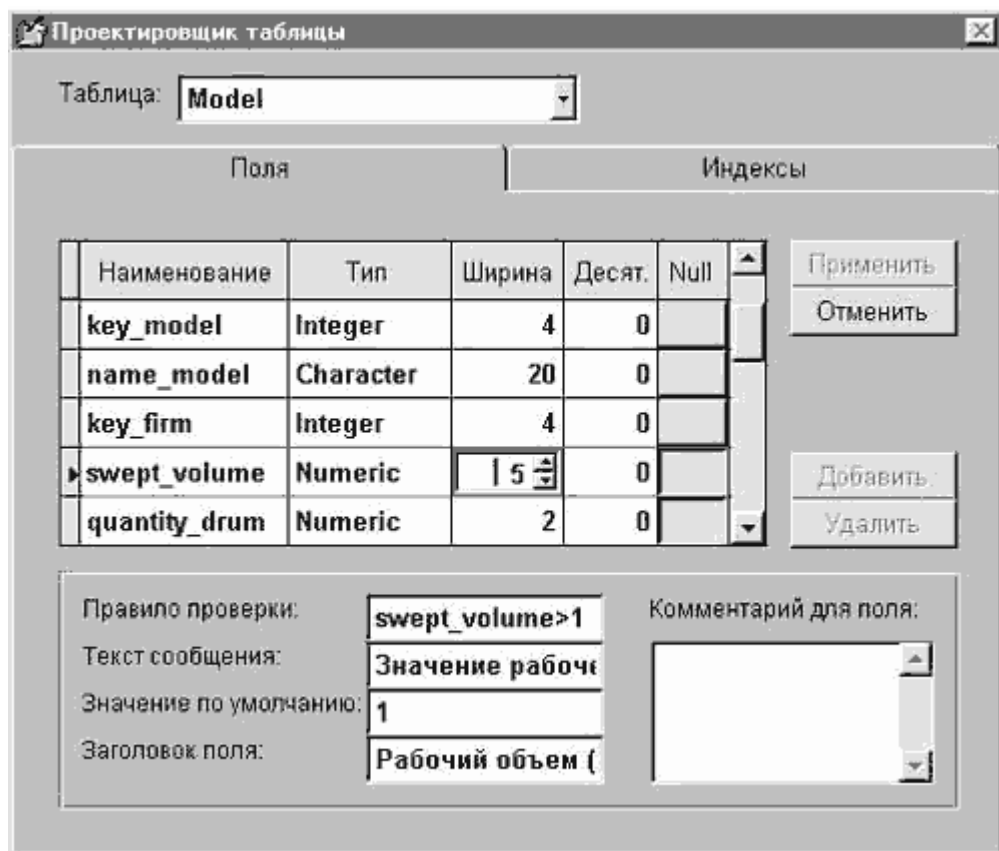


Рис. 2.22.

- Щелчок левой кнопки мыши по имени индекса под списком полей таблицы активизирует индекс данной таблицы.
- Двойной щелчок левой кнопки мыши по имени индекса под списком полей таблицы вызывает Конструктор таблицы на вкладке индексов (рис. 2.23).
- Щелчок левой кнопки мыши по связи между таблицами позволяет выделить данную связь.

Мы думаем, что после просмотра данного примера у вас появится желание создать собственный словарь данных и включить туда целый ряд дополнительных возможностей, например, для каждого поля таблицы включить определение класса, обеспечивающего работу с данными.

2.4. Администрирование базы данных

С базой данных, как правило, взаимодействуют несколько пользователей. Эти пользователи в организации могут выполнять совершенно различные функции, иметь различные представления об используемых данных, но пользоваться ими одновременно. Поэтому при эксплуатации БД крайне необходим учет различных требований и наличие алгоритма разрешения конфликтов.

Иными словами, нужно ввести долгосрочную функцию администрирования, направленную на координацию и выполнение всех этапов проектирования, реализации и ведения интегрированной базы данных. В соответствии с этой функцией на определенных лиц возлагается ответственность за сохранность такого важного ресурса, как данные.

В этом параграфе мы познакомимся с задачами и функциями администрирования БД, которое необходимо обеспечить на самых ранних стадиях разработки.

Характерное для многопользовательской среды использование хранящихся в компьютере данных заключается в "захватывании" файла данных. Каждый пользователь блокирует свои данные, не допуская остальных к их использованию. Это вынуждает других пользователей накапливать те же самые данные. С появлением баз данных необходимость в индивидуальном хранении и использовании информации отпала. Но появилась потребность в управлении коллективным использованием данных.

Администратором базы данных (АБД) называется лицо, ответственное за выполнение функции администрирования базы данных.

АБД - не "обладатель" базы данных, а ее "хранитель". С усложнением предметной области неизбежно усложняется процесс формирования информации и принятия решений. В результате расширяется спектр функций администрирования. Так как в случае использования базы данных прикладной программист "устраняется" от непосредственного управления данными, он утрачивает с ними контакт, а следовательно, и чувство ответственности за них. Это требует разработки процедур обеспечения непротиворечивости данных, которые должны быть скоординированы с функцией администрирования базы данных.

Администрирование базы данных предполагает обслуживание пользователей базы данных. Можно провести аналогию между АБД и ревизором предприятия. Ревизор защищает ресурсы предприятия, которые называются деньгами, а АБД - ресурсы, которые называются данными. Во многих организациях по странной традиции АБД рассматривают только как квалифицированного технического специалиста, часто совмещающего функции программиста. Это не соответствует целям администрирования. Уровень АБД в иерархии организации должен быть достаточно высоким, чтобы он мог определять структуру данных и право доступа к ним и нести за это ответственность. АБД обязан хорошо представлять себе, как работает предприятие и как оно использует данные. Хотя от АБД и требуется техническая компетентность, не менее важным является понимание им предметной области, а также умение общаться с людьми и подчинять альтернативы стандартным процедурам. В противном случае АБД не сможет эффективно выполнять свои функции.

Весьма заманчиво наделить АБД широкими полномочиями, однако его положение на предприятии может быть различным. Оно зависит прежде всего от степени значимости базы данных для жизнедеятельности данного предприятия. Вторым фактором является уровень сложности обработки данных и организации коммерческой деятельности. Как мы уже отмечали, АБД чаще всего назначается из числа прикладных программистов отдела обработки данных, что не всегда оправдано.

АБД должен координировать действия по сбору сведений, проектированию и эксплуатации базы данных, а также по обеспечению защиты данных. АБД обязан учитывать как перспективные, так и текущие информационные требования предметной области. Это одна из его главных задач. Следовательно, при проектировании базы данных необходимо добиваться ее максимальной гибкости или максимальной независимости данных.

Переход при обработке информации на технологию баз данных и расширение существующей базы данных связаны со значительными финансовыми затратами, что предопределяет необходимость тщательного планирования и управления этим процессом. Кроме того, количество данных, помещаемых в базу, растет день ото дня, и параллельно с этим усложняются обрабатываемые эти данные прикладные программы. Все это требует наличия централизованного управления на каждом этапе жизненного цикла системы с базой данных.

Правильная реализация функций администрирования БД существенно улучшает контроль и управление ресурсами данных предметной области. С этой точки зрения функции АБД являются скорее управляющими, чем техническими. Принципы работы АБД и его функции определяются подходом к данным как к ресурсам организации. Поэтому решение проблем, связанных с администрированием БД, часто начинается с установления общих принципов эксплуатации СУБД, хотя между СУБД и администрированием БД имеется различие. В большинстве случаев СУБД покупается в виде программного пакета, а администрирование БД является прерогативой предприятия. АБД обеспечивает обобщенное представление о предметной области в виде концептуальной модели, которая представляет модель данных предприятия. Одной из целей создания базы данных является обеспечение информацией пользователей, работающих в различных функциональных областях предприятия. Однако это часто означает, что ни один из этих пользователей не испытывает чувства ответственности и об общих интересах заботится в

последнюю очередь. Неизбежным результатом такого отношения является превращение АБД в координатора. Во многих случаях, организуя базу данных, АБД идет по пути решения технических проблем, то есть прежде всего вопросов, связанных с использованием СУБД.

Первая важная задача АБД состоит в устранении противоречий между различными направлениями деятельности организации при создании концептуальной, а затем и логической модели базы данных предметной области. Выступая в роли посредника между отделами, он должен добиваться не только того, чтобы различные специалисты пришли к соглашению относительно объектов предметной области, но и того, чтобы это соглашение было "правильным". Кроме определения данных и прав доступа к ним от АБД может потребоваться разработка процедур и руководств по ведению данных. Для выполнения функций АБД необходимо хорошо представлять себе состояние дел предприятия и перспективы его развития, а также знать позицию руководства. На начальной стадии разработки базы данных АБД следует сконцентрировать внимание на следующих проблемах:

- определение элементов данных и объектов предметной области;
- присвоение различных имен, которые будут использоваться для обращения к элементам одного и того же типа;
- установление взаимосвязей между элементами данных; выпуск текстового описания элементов данных;
- выделение отделов или пользователей, ответственных за обеспечение точности данных (например, контролирующих обновление данных, их непротиворечивость);
- определение путей применения элементов данных в целях управления и планирования, то есть распределении функций между персоналом.

Сбор всей этой информации из различных источников и необходимость ликвидации возникающих между отделами трений требуют, чтобы АБД обладал еще и дипломатическими способностями.

Как уже отмечалось, понятие "единоличного владения" данными неприменимо к базе данных. Однако оно существует, и это иногда существенно усложняет работу АБД. АБД приходится убеждать некоторые отделы, чтобы они "передали" свою собственность в общее пользование, либо контролировать доступ к легко искажаемой информации.

Идея "разделения" может не только вызвать противодействие со стороны некоторых отделов, но и настроить их враждебно против всего проекта разработки базы данных в целом. АБД должен одних убедить, других уговорить, третьих ободрить, а кого-то, если необходимо, и принудить. Это означает, что АБД должен уметь пользоваться своей властью и влиянием, обладать определенным стажем работы и хорошо разбираться в обстановке на данном предприятии. Очевидно, функции АБД не может исполнять человек, восстановивший за время работы против себя многих сотрудников.

Таким образом, к вопросу выбора АБД администрация предприятия должна подходить чрезвычайно серьезно. При выдвижении кандидатуры на пост АБД следует руководствоваться теми же критериями, что и при назначении на посты других управляющих, поскольку рассмотрению долговременных потребностей предприятия АБД обязан уделять не меньшее (если не большее) внимание, чем текущим проблемам. Выполнение этой обязанности осложняется еще и тем, что база данных предусматривает объединение данных без учета функциональных границ.

Реализация руководящих материалов может быть успешной только в том случае, когда все сотрудники, имеющие отношение к базе данных, ознакомлены с ними и несут ответственность за выполнение стандартов, устанавливаемых АБД. Прикладные программисты, сотрудники служб эксплуатации и сопровождения системы должны понимать процедуры, требуемые для решения стоящих перед ними задач. Это означает, что АБД необходимо установить эффективную взаимосвязь со всеми группами сотрудников, которым приходится обращаться к базе данных.

Возвращаясь к рассмотренной в конце предыдущего параграфа программе *Auto_Store*, заметим, что она позволяет: добавлять, удалять пользователей, назначать существующим пользователям имена, пароли, уровни доступа, назначать привилегии доступа относительно каждого уровня доступа. На рис. 2.24 и 2.25 изображены формы, соответствующие описанным функциям.

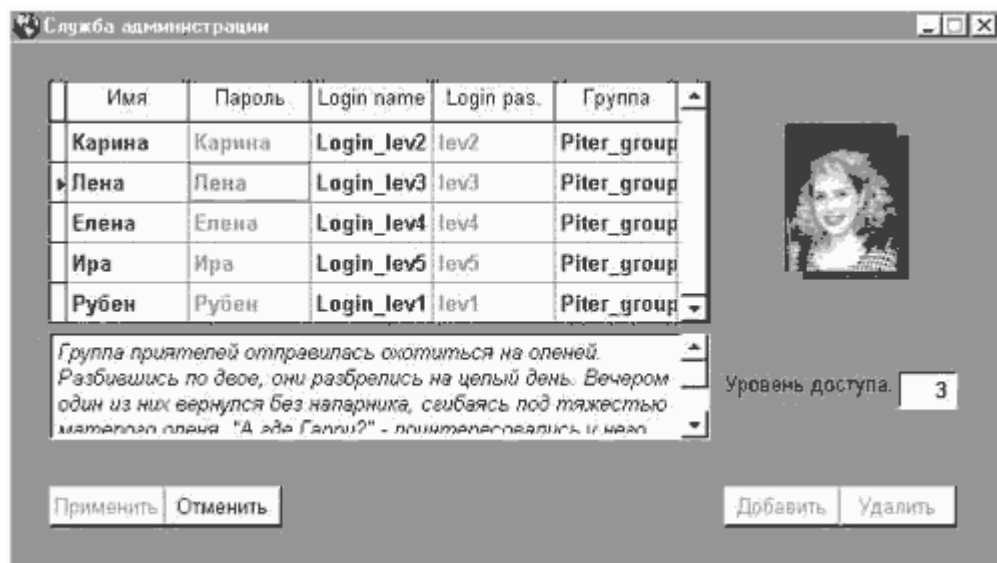


Рис. 2.24.

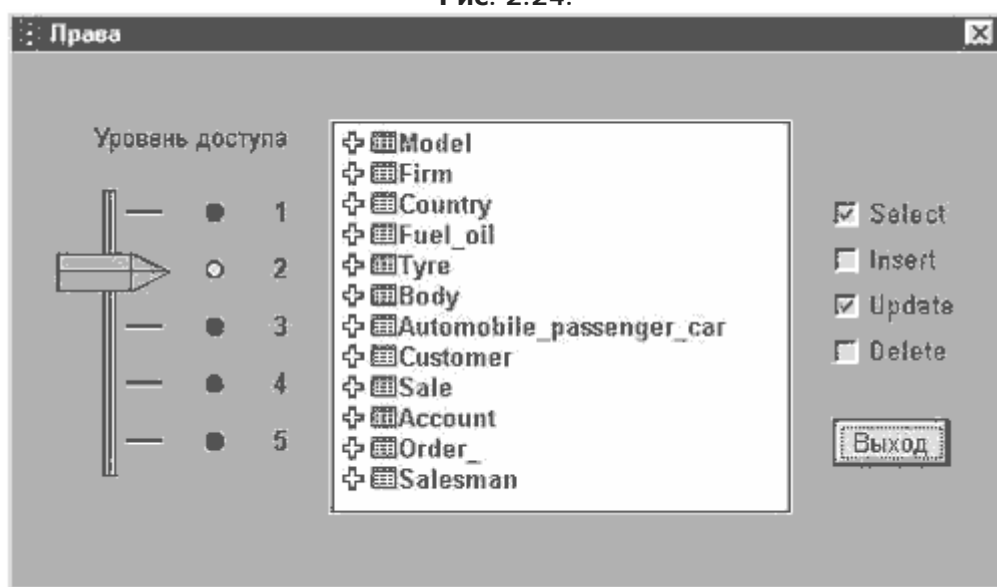


Рис. 2.25.

Кроме того, с помощью этой программы можно регистрировать пользователей для обращения к серверу, как это показано на рис. 2.26.



Рис. 2.26. Форма для соединения с SQL Server

Глава 3

Обзор возможностей и особенностей различных СУБД

3.1. Средства быстрой разработки приложений

3.2. Visual FoxPro

- Project Manager
- Database Designer
- Form Designer
- Visual Class Designer
- Query / View Designer
- Connection Designer
- Report/Label Designer
- Menu Designer

Вспомогательные средства разработчика

3.3. Access

- Запросы
- Формы
- Отчеты
- Макросы
- Система защиты

3.4. Visual Basic

3.5. MS SQL Server

3.6. Руководство для покупателя

После того как в предыдущих двух главах мы обсудили важнейшие вопросы теории построения баз данных, пора посмотреть на те средства, с помощью которых программист воплощает свои идеи в жизнь. В этой главе мы обсудим сильные и слабые стороны рассматриваемых в книге средств разработки. Для начинающих программистов мы предусмотрели небольшой обзор каждого продукта, что, как мы надеемся, существенно сократит сроки его освоения. Кроме того, как показал наш опыт общения с большим числом программистов, даже самые опытные из них часто слабо представляют себе комплектацию коммерческих средств разработки и возможности предлагаемого вспомогательного инструментария. Поэтому мы решили включить в эту главу материалы и на эту тему.

3.1. Средства быстрой разработки приложений

Фраза, вынесенная в заголовок этого параграфа, в англоязычной компьютерной литературе имеет очень лаконичную аббревиатуру - RAD (Rapid Application Development) и все чаще встречается на страницах специализированных изданий. Что это такое? Это очередной этап, причем этап революционный, развития информационных технологий. Естественная реакция компьютерной индустрии на информационные потребности быстроразвивающегося общества.

В этом параграфе мы изучим основные черты рассматриваемых в книге средств создания приложений для обработки данных и попытаемся их сравнить.

В мире уже используются десятки миллионов персональных компьютеров и их число постоянно растет. Компьютеры применяются в тех областях, где о них не помышляли еще год назад. Компьютеры начинают вытеснять даже такие, казалось бы, незыблемые атрибуты цивилизации, как телевизор и другую привычную нам бытовую технику. А увеличение числа и расширение сферы применения компьютеров ведет к увеличению потребности в программном обеспечении. Единственный путь, уводящий от необходимости превратить все трудоспособное человечество в программистов, - резкое повышение эффективности средств разработки программ. Эта идея и воплощается в современных версиях пакетов программ для создания систем автоматизации обработки данных, которые отвечают требованиям RAD. Можно выделить следующие отличительные черты таких средств разработки:

- Наличие объектно-ориентированного языка программирования, позволяющее очень эффективно использовать модульный принцип составления программ.
- Визуальные средства разработки, предоставляющие возможность заменить написание программного кода рисованием пользовательского интерфейса и заданием необходимой функциональности диалоговыми средствами.
- Поддержка стандартных протоколов обмена данными между приложениями, позволяющая разрабатывать многоуровневые приложения, не зависящие от источника данных. Здесь же заложена возможность применения компонентной технологии создания приложений.
- Возможность создания приложений клиент-сервер, позволяющая разрабатывать приложения неограниченной сложности и обеспечивать потребности целого предприятия в обработке данных.

Ни один строитель не построит дом быстрее малыша, складывающего его из кубиков. Задача современного средства разработки - дать нам много разных кубиков. Задача программиста - взять нужный кубик и поставить его в нужное место. Это основная идея RAD!

Перечень современных средств разработки систем автоматизации обработки данных, в которых заложены идеи RAD, весьма обширен. Почти каждый месяц появляются новые версии этих продуктов той или иной фирмы - производителя программного обеспечения. Они включают все новые и новые возможности, облегчающие труд программиста. В этой книге мы расскажем, как создать систему автоматизации обработки данных с помощью средств разработки Корпорации Microsoft. "Почему Microsoft?" - спросит пыливый читатель. Авторы все вместе и каждый по отдельности дали на это несколько ответов:

- Microsoft - самая крупная и на данный момент наиболее удачливая фирма-производитель программного обеспечения.
- Программами Microsoft пользуются десятки миллионов человек во всем мире.
- Лично я знаю и умею использовать только средства разработки Microsoft.
- Средства разработки Microsoft отлично интегрированы между собой, поддерживают все современные протоколы обмена данными, и поэтому всегда можно использовать наиболее эффективный в конкретной ситуации пакет программ.

В последующих главах на конкретных примерах мы покажем наиболее эффективные решения, которые можно реализовать с помощью рассматриваемых программ. А сейчас попробуем очертить сферу применения средств разработки Microsoft. Эта компания в настоящий момент предлагает пять пакетов программ, которые могут быть использованы для создания пользовательского приложения по обработке данных: Access, SQL Server, Visual Basic, Visual C++ и Visual FoxPro. Эти средства могут быть использованы как по отдельности - для решения конкретной поставленной задачи, так и в качестве интегрированного набора, каждый компонент которого может быть использован при разработке больших проектов масштаба предприятия. С этой точки зрения характеристика всех пяти продуктов приведена в табл. 3.1.

Таблица 3.1. Сравнение средств разработки Microsoft

Название	Основные	Основное
----------	----------	----------

продукта	преимущества	назначение
Access	Простота освоения. Возможность использования непрофессиональным программистом. Имеет мощные средства подготовки отчетов из БД различных форматов	Создание отчетов произвольной формы на основании различных данных. Разработка не коммерческих приложений.
SQL-Server	Высокая степень защиты данных. Мощные средства работы с данными. Высокая производительность	Хранение больших массивов данных. Хранение данных, требующих соблюдения режима секретности или при не допустимости их потери.
Visual Basic	Универсальность. Возможность создания компонентов OLE. Невысокие требования к мощности ПЭВМ	Создание приложений средней мощности, не связанных с большой интенсивностью обработки данных. Разработка компонентов OLE. Создание приложений для интеграции компонентов Microsoft Office.
Visual C++	Универсальность. Наибольшая скорость работы приложения. Неограниченная функциональность	Создание компонентов приложения для выполнения критичных по скорости процессов или обеспечения функциональности, не достижимой в других средствах разработки.
Visual FoxPro	Высокий уровень объектной модели. Высокая скорость обработки данных. Интеграция объектно-ориентированного языка программирования с Xbase и SQL. Многоплатформенность	Создание приложений масштаба предприятия. Создание приложений для работы на различных платформах (Windows 3.x,, Windows 95,, Macintosh и т. д.).

Рассмотрим теперь более подробно перечисленные в табл. 3.1 средства разработки за исключением пакета Visual C++, который, являясь инструментом профессионала, даже для краткого рассказа потребует книги более объемной, чем лежащая перед вами.

Какие общие черты имеют рассматриваемые средства разработки, подтверждающие наше утверждение о возможности их совместного использования для разработки пользовательских

приложений различного уровня сложности? Во-первых такие новые технологии, как OLE, ODBC, DAO, RDAO, ActiveX и пр., которые они поддерживают. В этой книге мы постарались обратить на них самое пристальное ваше внимание. Эти технологии закладывают возможность использования в одном приложении данных, хранящихся в различных форматах. Мы можем легко разрабатывать приложения, независимые от данных. Помимо этого, за счет OLE Automation, мы можем использовать функциональные возможности различных пакетов программ для выполнения с данными специфических операций. Классическим примером такой возможности является подготовка в приложении, написанном на Visual FoxPro, данных, хранящихся в формате Access, для вывода в виде сложного графика с использованием Мастера подготовки графиков Excel.

Конечно, при совместном использовании различных средств разработки приложений нас больше всего будут интересовать данные. В табл. 3.2 приведен перечень типов данных, доступных в рассматриваемых средствах разработки. Прочерки в двух предпоследних колонках таблицы обозначают, что для этого типа данных задание конкретных величин не требуется.

Таблица 3.2. Типы данных

Тип данных	Visual FoxPro	Access и Visual Basic	MS SQL Server	Длина	Число десятичных разрядов	Занимаемый объем
Binary Image	Нет	dbLongBinary	binary(n)	n байт	-	до 1,2 Гбайт
Byte	Нет	dbByte	tinyint	1	-	1 байт
Character Text	C	dbText	char(n), varchar(n)	n	-	4 байта
Count	Нет	dbLong	Нет	-	-	4 байта
Currency	Y	dbCurrency	money	-	-	8 байт
Date	D	Нет	Нет	-	-	8 байт
DateTime	T	dbDate	datetime	-	-	8 байт
Logical (Yes/No)	L	dbBoolean	bit	-	-	1 байт
Numeric	N	Нет	float	n	d	от 1 до 20 байтов
Integer	Нет	dbInteger	smallint	-	-	2 байта
Integer	I	dbLong	int	n	-	4 байта
Double	B	dbDouble	float	-	d	8 байт
Float	F		float	n	-	от 1 до 20 байтов
General (OLE Object)	G	dbLong-Binary	image	-	-	4 байта
Memo	M	dbMemo	text	-	-	4 байта
Single	Нет	dbSingle	real	-	-	4 байта
Character (binary)	C	Нет	Нет	n	-	1 байт на символ
Memo (binary)	M	Нет	Нет	-	-	4 байта

- **Binary Image.** Любые данные в двоичном виде. Используется для хранения изображений, файлов и т. д.
- **Byte.** Целое положительное число от 0 до 255.
- **Character.** Символьное выражение может содержать любые символы (до 254 для одного поля).
- **Count.** Счетчик, который автоматически наращивает свое значение при добавлении записи. Начальное значение 1.
- **Currency.** Денежное выражение для числовой величины. Выводит число с четырьмя десятичными разрядами и установленным обозначением используемой денежной единицы.
- **Date.** Выражение для даты может содержать день, месяц и год.
- **DateTime.** Выражение дата и время может содержать время, день, месяц и год.
- **Logical.** Булево выражение для .T. или .F..

- **Numeric.** Числовое выражение может содержать целые или дробные числа со знаком.
- **Integer (dbInteger).** Целое число в диапазоне от -32,768 до +32,767.
- **Integer (dbLong).** Целое число. Можно хранить числа от -2147483647 до 2147483646.
- **Double.** Числа с плавающей точкой двойной точности. Можно хранить значения от 4.94065645841247E-324 до 1.79769313486232E308.
- **Float.** То же, что числовое выражение. Оставлено для совместимости.
- **General.** Поле для ссылки на объект OLE.
- **Memo.** Поле примечаний для ссылки на блок данных.
- **Single.** Число с плавающей точкой одинарной точности. Можно хранить отрицательные числа от -3.402823E38 до 1.401298E-45 и положительные числа от 1.401298E-45 до 3.402823E38.
- **Character (binary).** Символьное выражение, не подвергаемое трансляции в другую кодовую страницу.
- **Memo (binary).** Поле примечаний для ссылки на блок данных, не подвергаемых трансляции в другую кодовую страницу.

Все СУБД, как правило, имеют сходный функциональный состав, в который входят диалоговые средства для работы с данными - назовем их пользовательскими средствами, средства разработчика, обеспечивающие возможность создания пользовательского приложения, и дополнительные средства, от состава которых, как правило, зависят функциональные возможности и мощность разрабатываемых программ. Отражающая такой подход функциональная схема интерфейса СУБД представлена на рис. 3.1. В зависимости от назначения средства разработки, о чем мы уже говорили ранее, состав различных средств в конкретной СУБД может значительно отличаться. Например, в Access пользовательские средства развиты значительно сильнее, чем в Visual Basic, где они рассматриваются как вспомогательные функции.

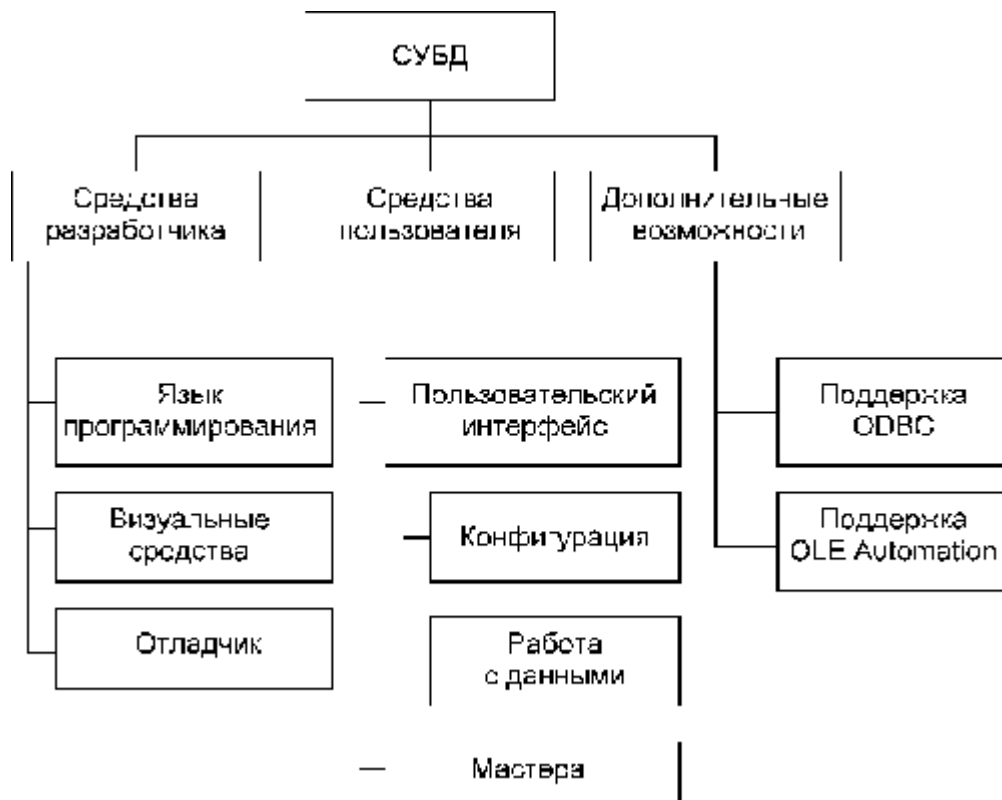


Рис. 3.1. Функциональная схема интерфейса СУБД

3.2. Visual FoxPro

Visual FoxPro - не просто следующая версия одной из наиболее быстрых СУБД для персональных компьютеров. Это совершенно новая программа, которая позволяет легко сделать то, что в предыдущих версиях давалось с величайшим трудом или было просто недоступно. Главное окно Visual FoxPro приведено на рис. 3.3.



Рис. 3.3.

Интерфейс Visual FoxPro отвечает представлениям о современной графической среде; напоминая интерфейс иных программ Microsoft, делает работу интуитивно понятной. Основная работа с данными в Visual FoxPro выполняется с помощью различных инструментальных средств, поэтому команды меню часто имеют вспомогательный характер и их состав гибко меняется в зависимости от того, какое средство активно в данный момент. Дадим краткую характеристику основным командам меню.

Меню **File** включает основные функции для работы с файлами:

- **New** - создает новый файл. Тип создаваемого файла можно выбрать из появляющегося диалогового окна, представленного на рис. 3.4.

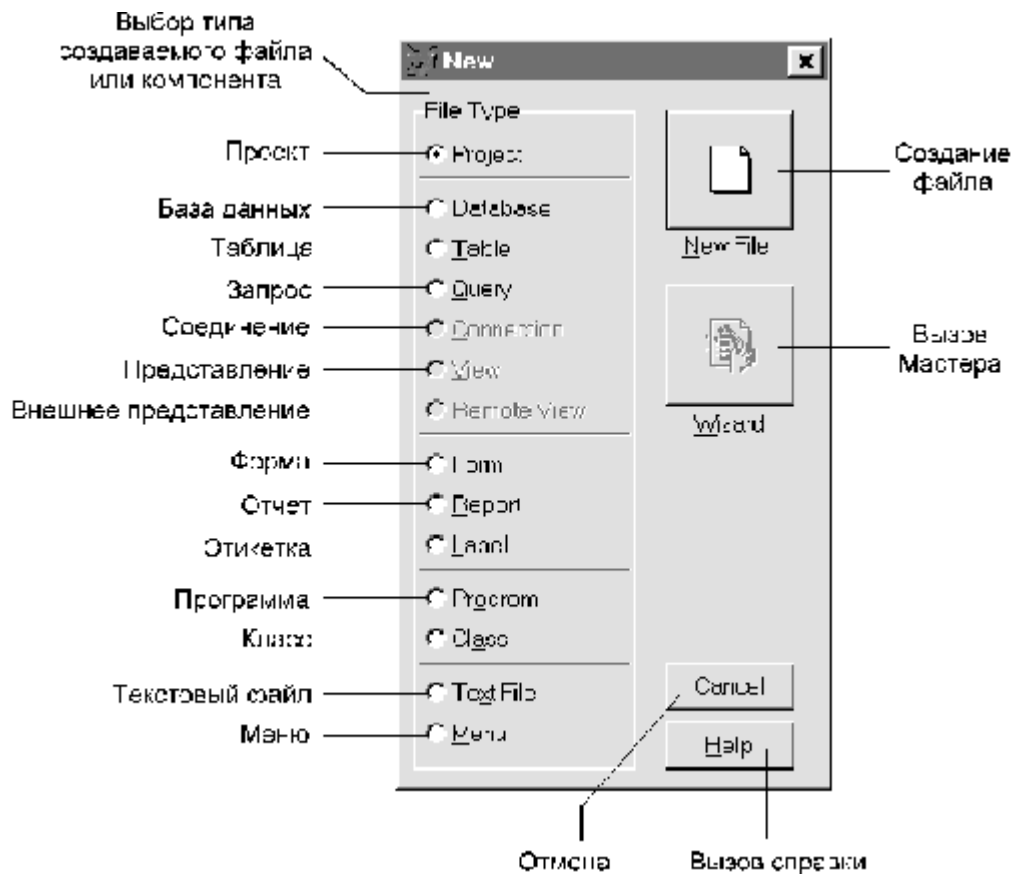


Рис. 3.4.

- **Open** - открывает существующий файл.
- **Close** - закрывает активное окно. Если вы удерживаете клавишу **Shift**, закрываются все открытые окна.
- **Save** - сохраняет изменения, сделанные в активном файле.
- **Save As** - сохраняет активный файл с другим именем.
- **Revert** - отменяет все изменения, сделанные в активном файле с момента последнего сохранения.
- **Import** - позволяет импортировать данные в таблицу Visual FoxPro.
- **Export** - позволяет экспортировать данные из Visual FoxPro в другие форматы.
- **Page Setup** - вызывает диалоговое окно для установки параметров страницы отчета или этикетки.
- **Print Preview** - позволяет просмотреть на экране результат выполнения отчета или этикетки.
- **Print** - выводит на печать или в файл текстовый файл, программу, отчет, этикетку, содержимое окна **Command** или буфера обмена (Clipboard).
- **Send** - позволяет отправить электронную почту при наличии на компьютере соответствующих средств.
- **Exit** - закрывает Visual FoxPro.

Меню **Edit** позволяет выполнять редактирование программного кода или любого другого текста, а также облегчает работу с элементами управления и объектами в формах, отчетах и т. д.

- **Undo** - отменяет последнее выполненное действие.
- **Redo** - восстанавливает последнее отмененное действие.
- **Cut** - удаляет и записывает в буфер обмена выделенный фрагмент или элементы управления.
- **Copy** - записывает в буфер обмена выделенный фрагмент или элементы управления.
- **Paste** - копирует из буфера обмена хранящиеся там данные в место расположения курсора.
- **Paste Special** - связывает или встраивает из буфера обмена хранящиеся там OLE-объекты в место расположения курсора.
- **Clear** - удаляет выделенный фрагмент или элементы управления.

- **Select All** - выделяет весь текст или элементы управления в активном окне.
- **Find** - выводит диалоговое окно поиска фрагмента текста.
- **Replace** - выводит диалоговое окно поиска фрагмента текста и его замены на другой фрагмент.
- **Go To Line** - выполняет быстрый переход на указанный номер строки в текстовом файле.
- **Insert Object** - выводит диалоговое окно со списком объектов, которые можно встроить в активную форму или редактируемое поле типа **General**.
- **Object** - выводит список действий, поддерживаемых активным объектом (например, при просмотре содержимого поля типа **General**).
- **Links** - выводит диалоговое окно для редактирования или удаления связи с активным объектом

Меню **View** позволяет управлять появлением на экране вспомогательных средств, например панелей инструментов, и выводить на экран данные. При отсутствии на экране каких-либо средств разработки в этом меню доступна только одна команда - **Toolbars**, которая выводит диалоговое окно для выбора размещаемых на экране панелей инструментов.

Меню **Format** появляется всегда, когда на экране есть активное окно, и позволяет изменять внешний вид отображаемых данных с помощью следующих команд:

- **Font** - выводит диалоговое окно изменения шрифта и его характеристик.
- **Enlarge Font** - увеличивает размер отображаемого в активном окне текста.
- **Reduce Font** - уменьшает размер отображаемого в активном окне текста.
- **Single Space** - устанавливает один межстрочный интервал для отображаемого в активном окне текста.
- **1 1/2 Space** - устанавливает полуторный межстрочный интервал для отображаемого в активном окне текста.
- **Double Space** - устанавливает двойной межстрочный интервал для отображаемого в активном окне текста.
- **Indent** - позволяет сдвинуть вправо линию или несколько линий текста на один интервал табуляции в окне редактора или окне **Command**.
- **Remove Indent** - позволяет сдвинуть влево линию или несколько линий текста на один интервал табуляции в окне редактора или окне **Command**.

Меню **Tools** позволяет выполнить различные вспомогательные действия:

- **Wizards** - запускает один из имеющихся Мастеров.
- **Spelling** - запускает программу проверки правописания для содержимого текстового файла или поля примечаний (можно использовать для проверки правильности написания команд).
- **Macros** - позволяет присвоить клавиатурной комбинации выполнение какого-либо действия или набора действий.
- **Class Browser** - выводит на экран утилиту работы с классами.
- **Trace Window** - выводит на экран окно для визуального отображения выполняемого программного кода.
- **Debug Window** - выводит на экран окно для отображения текущих значений в процессе выполнения программы.
- **Options** - выводит на экран диалоговое окно для установки параметров конфигурации среды разработки.

Меню **Program** содержит команды, связанные с выполнением программ:

- **Do** - запускает программу на выполнение.
- **Cancel** - заканчивает выполнение
- **Resume** - продолжает выполнение программы со строки, на которой она была приостановлена командой **Suspend**.
- **Suspend** - приостанавливает выполнение программы без выгрузки ее из памяти, оставляя возможным продолжение ее работы командой **Resume**.
- **Compile** - компилирует программу в псевдокод. Меню **Window** содержит команды управления окнами:
- **Arrange All** - располагает все открытые окна на экране так, чтобы каждое было видимо.
- **Hide** - скрывает активное окно.
- **Clear** - стирает содержимое активного окна.

- **Circle** - выполняет переход к следующему открытому окну.
- **Command Window** - делает активным или открывает окно **Command**. Среди рассматриваемых средств разработки это окно является уникальным, так как позволяет немедленно выполнять почти что все команды **Visual FoxPro** и, соответственно, видеть результат их работы.
- **View Window** - делает активным или открывает диалоговое окно **View**, которое содержит основной инструментарий для работы с данными.

Меню **Help** содержит команды, которые позволяют быстро получить необходимую информацию о работе с **Visual FoxPro**.

Отличительные черты **Visual FoxPro** можно описать следующим образом:

1. Обеспечение возможности быстрой разработки прикладной программы базируется на включении средств, которые позволяют повысить скорость работы программиста. В первую очередь это средства объектно-ориентированного программирования, позволяющие пользователю формировать компоненты своего проекта (объекты), которые затем могут многократно использоваться. В связи с этим, традиционный **Xbase** язык в **Visual FoxPro 3.0** значительно расширен, что позволяет создавать истинные объекты, классы и подклассы. Кроме того, объекты могут быть созданы с помощью визуальных средств и многократно использоваться в любое время.
2. Обеспечение полного набора средств для управления событиями. Традиционно в **Xbase** от программиста требовалось написать собственный драйвер для обработки необходимого набора событий или положиться на **READ**-состояние ожидания, которое моделирует обработку события системой. В **Windows** число событий, к которым может обращаться пользователь, весьма велико, и, следовательно, обработка событий является непростой задачей. **Visual FoxPro 3.0** имеет истинно управляемую событиями модель, так что по умолчанию система раньше, чем пользователи, обрабатывает объектные события. Кроме того, программист теперь имеет полный доступ к набору стандартных, основанных на функционировании **Windows** событий (например, движения мыши, которые допускают перетаскивание объектов).
3. Обеспечение мощного набора инструментальных средств для программиста. Разработчики систем автоматизации обработки данных кроме мощного набора визуальных средств проектирования могут использовать широкие возможности по интеграции систем хранения данных и доступа к серверам данных с помощью технологии **ODBC**. Основные новшества - это расширение встроенного языка **SQL**, возможность обновления данных на сервере через редактирование курсоров, встроенный механизм обеспечения транзакций, возможность обращения к серверу на том диалекте **SQL**, который поддерживает сервер. Наличие словаря данных делает более быстрой разработку структуры баз данных и облегчает ее дальнейшую эксплуатацию и поддержку.
4. Обеспечение полной интеграции **Visual FoxPro 3.0** в семейство прикладных программ **Microsoft**. Единый интерфейс с наиболее популярными прикладными программами **Microsoft** делает работу в интерактивном режиме интуитивно понятной. Поддержка правой кнопки мыши позволяет избежать долгих путешествий по системе меню и значительно облегчает изучение новых возможностей СУБД. Просто выберите курсором объект и нажмите правую кнопку мыши! На некоторых диалоговых окнах, которые часто используются в работе на полосе заголовка, появился переключатель в виде анимационной пиктограммы (**push pin**), позволяющий легко включить режим, при котором это окно будет всегда расположено на переднем плане. **Visual FoxPro** обеспечивает полную поддержку **OLE 2.0**, что облегчает взаимодействие с другим программным обеспечением в среде **Windows**. Помимо оставшейся возможности загрузки внешних функций посредством команды **SET LIBRARY** появилась возможность обращения к функциям динамических **DLL** библиотек **Windows** посредством команды **DECLARE**.
5. Совместимость с ранее разработанным программным обеспечением в среде **FoxPro**.

В **Visual FoxPro** система организации данных наиболее близка к теоретическим основам реляционной модели и позволяет более естественно выполнять операции реляционной алгебры.

Основная единица хранения данных - это таблица, в столбцах и строках которой хранятся данные, как это и было раньше в **DBF**-файле. Таблица сохранила расширение файла **DBF** и имеет прямую совместимость со "старыми" **DBF**-файлами. Таблицы объединяются в базу данных, в которой можно описать все связи, устанавливаемые между полями отдельных таблиц, правила проверки, которые будут определять реакцию системы на вносимые изменения, добавление или удаление данных и правила проверки целостности данных в БД. Файлы баз данных имеют расширение **DBC** и при открытии автоматически поддерживают все перечисленные установки для входящих в нее таблиц. При необходимости можно иметь и таблицы, не входящие в БД, - свободные таблицы.

Visual FoxPro обеспечивает поддержку значений NULL и выполнение операций с этими данными в соответствии со стандартом ANSI. Это облегчает задачу представления неизвестных данных и взаимодействие с MS Access и базами данных SQL, которые могут содержать такие типы значений.

Таким образом, база данных в Visual FoxPro - это основной элемент организации данных, который, помимо формирования структуры представления информации, выполняет функции словаря данных за счет поддержки следующих функциональных возможностей:

- Допускаются длинные имена таблиц.
- Каждому полю и таблице можно давать комментарии.
- Допускается использование длинных имен полей.
- Для полей помимо идентификаторов можно использовать заголовки, которые могут использоваться как в окне Browse, так и в качестве заголовков для колонок в объекте Grid.
- Введены значения по умолчанию для полей.
- Предусмотрены правила проверки для полей и записей при изменении и вводе новых данных.
- Имеются триггеры для поддержания целостности данных.
- Поддерживаются постоянные связи между таблицами, размещенными в БД.
- Имеются процедуры БД для описания сложных условий правил проверки.
- Можно использовать соединения для связи с внешними источниками данных.
- Поддерживаются локальные и внешние просмотры.

Основным средством редактирования данных в оболочке FoxPro являются полноэкранные средства Browse и Edit (Change). Browse позволяет редактировать данные в наиболее привычном для пользователя виде - табличном, а Edit - в виде колонки полей. Для просмотра данных из таблицы, открытой в какой-либо рабочей области, открывается отдельное окно. Для просмотра или редактирования данных в таблице достаточно открыть нужную таблицу и в меню View выбрать команду Browse. Вид открывающегося при этом окна Browse показан на рис. 3.5.

Маркер текущей записи

Кнопка вызова
системного меню

Price		
Название пакета	Ид носителя	Стоим
MS-DOS 6.22	35" или 5 25"	
MS-DOS 6.22 Bus L Upgrade	35" или 5 25"	
MS-DOS 6.22 Bus HIP		
MS-DOS 6.22 Bus Upgrade from 6.010 pack	35"	
Windows 3.11	35" или 5 25"	
Windows 3.11 disc	35" или 5 25"	
Windows 3.11 disc Upgrade	35" или 5 25"	
Windows for Workgroups 3.11	35" или 5 25"	
Windows for Workgroups 3.11 5 licens pack1	35" или 5 25"	
Windows for Workgroups 3.11 Add-on for MS-DOS	35" или 5 25"	
Windows for Workgroups 3.11 Add-on	35" или 5 25"	
Windows for Workgroups 3.11 Add-on Filter pack1	35" или 5 25"	

Маркер деления окна на части

Рис. 3.5. Окно Browse

Этот способ визуализации данных очень удобен, так как позволяет просматривать сразу несколько записей, но, как правило, редко когда все поля таблицы помещаются одновременно в окне, даже раскрытом на весь экран.

Просмотр данных в виде Edit позволяет работать сразу со всеми данными в одной записи, как это видно на рис. 3.6.

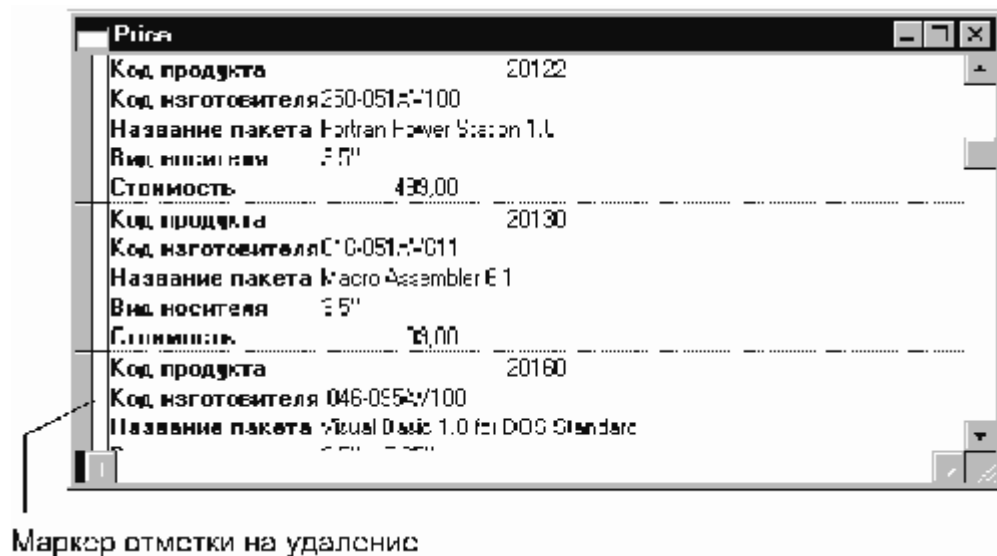


Рис. 3.6.

Переключаться между этими двумя видами просмотра данных можно с помощью соответствующих команд в меню View. При этом работа с данными не прерывается, меняется только вид представления информации. Обратите внимание, что, как видно на рис. 3.5 и 3.6, если в таблице описаны заголовки для полей, они используются в качестве идентификаторов. На этих же рисунках даны необходимые пояснения по управлению окном просмотра с помощью мыши.

Для ввода и редактирования данных могут использоваться приемы, обычно применяемые при работе с данными в программах для Windows. Используйте возможности, предлагаемые в меню Edit. Нажатие на клавишу **Tab** или **Enter** приводит к перемещению курсора в следующее поле, а для возврата в предыдущее удобно использовать сочетание клавиш **Shift + Tab**. Учтите, что при достижении курсором последнего символа в поле при вводе данных Visual FoxPro по умолчанию подает звуковой сигнал и переводит курсор в следующее поле.

Окна Browse или Edit являются мощными средствами просмотра и редактирования данных. Дополнительные возможности для достижения наивысшего удобства работы заложены в меню Table. Что предлагают нам команды этого меню?

- **Properties** - выводит на экран диалоговое окно, позволяющее установить характеристики для таблицы, открытой в данной рабочей области.
- **Font** - вызывает стандартное диалоговое окно Windows, которое позволяет выбрать удобный шрифт и подобрать его характеристики.
- **Go to Record** - позволяет быстро перейти к нужной записи, выбрав одну из следующих опций: **Top** - на первую запись, **Bottom** - на последнюю, **Next** - на следующую после текущей запись, **Previous** - на предыдущую после текущей, **Record#** - на запись с указанным номером. Не забудьте, что данные в окне просмотра располагаются в порядке их номеров, только если вы не используете какой-либо индекс. Опция **Locate** позволяет найти требуемую запись по ее содержанию, указав соответствующее выражение для поиска.
- **Append New Record** - добавляет в таблицу одну новую запись.
- **Toggle Deletion Mark** - помечает для удаления текущую запись или убирает эту отметку, если текущая запись уже помечена для удаления.
- **Append Records** - позволяет перейти в режим добавления записей, при котором новая запись будет автоматически добавляться после ввода данных в текущую. Добавленная запись будет сохранена, если вы ввели в нее хотя бы один символ. Мы настоятельно рекомендуем добавлять данные в таблицу именно этим методом, так как он позволяет избежать появления в таблице большого числа ненужных пустых записей.
- **Delete Records** - выводит диалоговое окно, позволяющее указать записи, которые необходимо пометить для удаления.
- **Recall Records** - выводит диалоговое окно, позволяющее указать записи, в которых необходимо убрать пометку для удаления.
- **Remove Deleted Record** - запускает команду **PACK** для физического удаления помеченных для этого записей.
- **Replace Field** - позволяет указать записи, данные в которых нужно заменить на указанное значение.

- **Size Field** - дает возможность изменить ширину колонки в окне для вывода данных из текущего поля. При этом вы изменяете ширину колонки, а не длину поля в таблице. При достижении нужной ширины колонки необходимо нажать клавишу **Enter**. Если ширина колонки меньше, чем длина поля в таблице, данные будут прокручиваться при перемещении курсора внутри колонки.
- **Move Field** - позволяет переместить текущую колонку в окне.
- **Resize Partitions** - позволяет изменить размеры частей окна просмотра или разбить это окно на две части, если этого не было сделано ранее. Разбиение окна позволяет оставить на экране какое-либо поле или поля при горизонтальном прокручивании данных. Это очень удобно при работе с таблицами, которые имеют большое количество полей. Пример разбиения окна **Browse** приведен на рис. 3.7. Мы можем установить различные виды просмотра данных в отдельных частях окна. Для этого достаточно перейти в нужную часть окна и в меню **View** выбрать соответствующую команду. Для больших таблиц можно рекомендовать организовать окно так, как это показано на рис. 3.8. В левой части окна просмотр данных установлен в стиле **Browse** и сюда помещена колонка с ключевыми данными для быстрого поиска нужных данных. Правая часть окна организована в стиле **Edit**, что позволяет видеть данные сразу из всех полей нужной записи.

Название пакета	Код продукта	Вид носителя	Стоимость
MS-DOS 6.22	10101	3.5" или 5.25"	66.00
MS-DOS 6.22 Rus Upgrade	10102	3.5" или 5.25"	66.00
MS-DOS 6.22 Rus MLP	10104		45.00
MS-DOS 6.22 Rus Upgrade from 6.0 10 pack	10103	3.5"	99.00
Windows 3.11	10110	3.5" или 5.25"	95.00
Windows 3.1 рус.	10215	3.5" или 5.25"	95.00
Windows 3.1 рус. Upgrade	10216	3.5" или 5.25"	75.00
Windows for Workgroups 3.11	10220	3.5" или 5.25"	102.00
Windows for Workgroups 3.1 (5 licens pack)	10225	3.5" и 5.25"	0.00
Windows for Workgroups 3.11 Add-on for MS	10224	3.5" и 5.25"	39.00
Windows for Workgroups 3.11 Add-on	10223	3.5" или 5.25"	39.00
Windows for Workgroups 3.1 Add-on (5 licens	10226	3.5" и 5.25"	399.00

Маркер изменения
размеров частей окна

Активная часть окна

Рис. 3.7.

Название пакета	Код продукта	Код изготовителя	Название пакета	Вид носителя	Стоимость
Windows NT Server 3.5 Upgrade	20122	250-051AV100	Fortran Power Station 1.0	3.5"	499.00
MS Source Profiler 1.0					
Fortran 5.1					
Fortran Power Station 1.0					
Macro Assembler 6.1	20130	016-051AV611	Macro Assembler 6.1	3.5"	109.00
Visual Basic 1.0 for DOS Standard					
Visual Basic 1.0 for DOS Pro					
Visual Basic 1.0 for DOS Pro Upgrade					
Visual Basic 3.0 Pro for Windows					
Visual Basic 3.0 Pro for Windows & C					
Visual C++ 1.0 Pro	20160	046-095AV100	Visual Basic 1.0 for DOS Stai	3.5" и 5.25"	
Visual C++ 1.5 Pro					

Рис. 3.8. Комбинирование двух видов представления данных в одном окне

- **Link Partitions** - позволяет синхронизировать перемещение по записям таблицы независимо от того, в какой части окна мы перемещаем записи. При отмене этой триггерной команды в каждой части окна просмотра записи будут перемещаться независимо.
- **Change Partitions** - обеспечивает переход из одной части окна в другую без использования мышки.
- **Rebuild Indexes** - позволяет в случае необходимости привести индексы в соответствие данным в таблице.

Для работы с данными, расположенными в полях примечаний, достаточно два раза щелкнуть мышкой в нужной ячейке окна или переместить туда курсор и нажать клавиши **Ctrl + PgDn**.

Visual FoxPro 3.0 продолжает поддерживать стандартное процедурное программирование Xbase, и при желании мы можем не обращать внимания ни на какую "объектность", однако новые расширения языка дают пользователям мощносты и гибкость объектно-ориентированного программирования, что предопределяет всю логику и методику разработки прикладной программы.

Вместо того чтобы ломать голову над программой, начиная с первой строки кода, программист может и должен думать о создании объектов - компонентов прикладной программы.

Классы и объекты - два фундаментальных понятия объектно-ориентированного программирования. Класс содержит информацию о том, как объект должен выглядеть и вести себя. Другими словами, класс - это прообраз объекта. Visual FoxPro 3.0 дает возможность пользователям создавать объекты как с помощью визуальных средств, так и программно на основании базовых классов

Для описания объекта используется набор свойств. Эти свойства объект получает из соответствующего класса, на основании которого он создан. Если нам нужен объект, имеющий свойства, отличные от свойства его класса, мы должны создать подкласс с измененными свойствами и уже его использовать для создания объекта.

Для описаний действий, выполняемых объектом, используются методы, то есть процедуры и функции, объявленные внутри класса и непосредственно с ним связанные. Методы легко координируются с событиями, происходящими при работе программы. Главное преимущество для разработчика здесь заключается в том, что привязанные к событию методы выполняются автоматически и у нас даже есть возможность принудительно вызвать какое-то событие. Для каждого класса список событий может меняться как в сторону расширения, так и сужения; более подробная информация по этому вопросу приведена в [пятой главе](#).

Прикладная программа, разработанная в СУБД FoxPro, может иметь достаточно сложную структуру и включать значительное количество файлов различного типа. Основные типы файлов Visual FoxPro приведены в табл. 3.3.

Таблица 3.3. Типы основных файлов в Visual FoxPro

Тип файла	Расширение файла	Расширение файла после компиляции
Пользовательское приложение, включающее в себя отдельные программные файлы	-	APP
База данных	DBC	-
Поля примечаний в БД	DCT	-
Индексный файл БД	DCX	-
Таблица	DBF	-
Индекс	IDX	-
Составной индекс	CDX	-
Поля примечаний таблицы	FPT	-
Текстовый файл с сообщениями об ошибках компиляции	ERR	-
Выполняемая программа, создаваемая на основе файла-приложения APP	-	EXE
Файлы макрокоманд	FKY	-
Отчет	FRX	-
Поля примечаний отчета	FRT	-
Программа	PRG	FXP
Этикетка	LBX	-
Поля примечаний этикетки	LBT	-
Поля примечаний меню	MNT	-
Меню	MNX	-
Сгенерированная программа меню	MPR	MPX

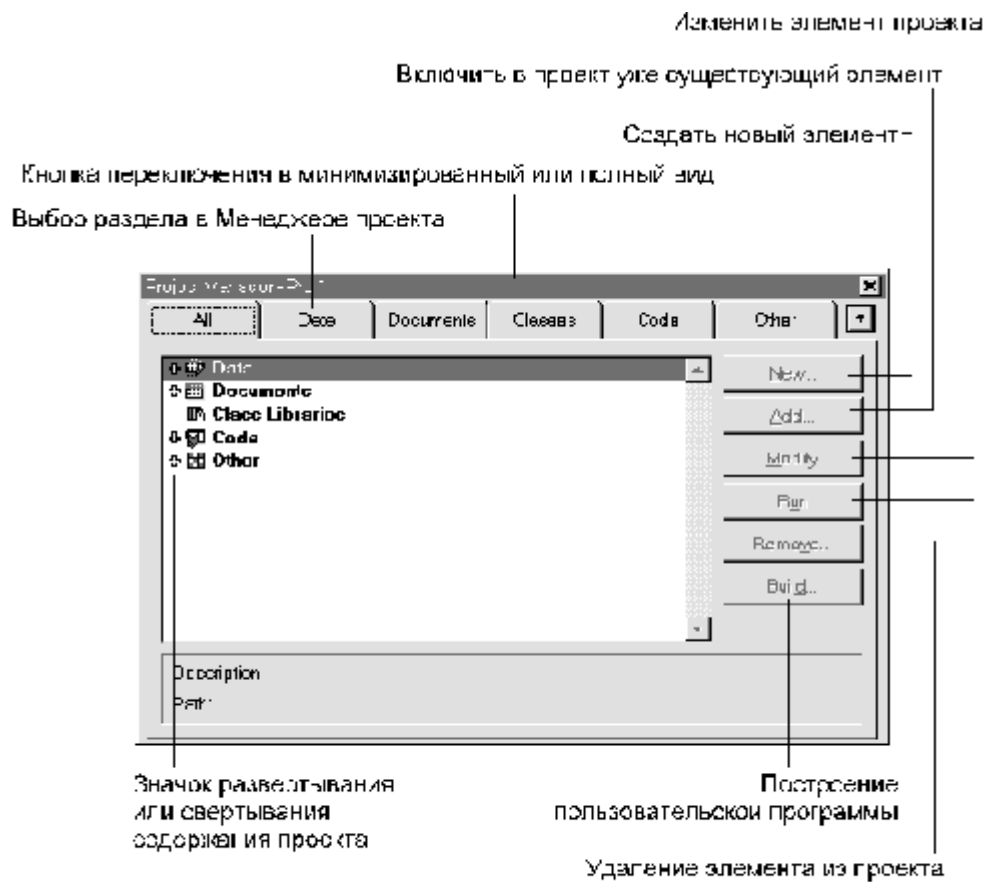
Файл элементов управления ActiveX	-	OCX
Проект	PJX	-
Поля примечаний проекта	PJT	-
Сгенерированный файл запроса	QPR	QPRX
Форма	SCX	-
Поля примечаний формы	SCT	-
Текстовые файлы	TXT	-
Визуальная библиотека классов	VCX	-
Поля примечаний визуальной библиотеки классов	VCT	-
Файл конфигурации Visual FoxPro	FPW	-

К радости разработчика большинство перечисленных файлов легко создается с помощью визуальных средств и, в частности, соответствующих Мастеров.

В процессе совершенствования FoxPro одним из ключевых моментов являлось постепенное развитие визуальных методов разработки пользовательских программ. В Visual FoxPro компанией Microsoft был расширен набор инструментальных средств для визуальной разработки. Как было упомянуто выше, одно из главных преимуществ Visual FoxPro 3.0 - увеличившаяся производительность разработчика, которая прежде всего и выражается в проектировании прикладных программ с помощью визуальных средств. Дадим краткую характеристику визуальных средств проектирования, включенных в версию 3.0.

Project Manager

Это центральный узел разработки прикладной программы, используемый для организации и управления файлами в проектах. Проект - это совокупность файлов, данных, документов и объектов FoxPro, информация о которых сохраняется в едином файле с расширением PJX. Возможности и структура Project Manager приведены на рис. 3.9.



Запуск исполняемого элемента проекта (программы, формы и т. д.), закрытие ЕД или таблиц, предварительный просмотр отчета — действие этой кнопки зависит от выбранного элемента проекта

Рис. 3.9. Возможности и структура Project Manager

Структурные единицы в Project Manager организованы в иерархической структуре просмотра, которую пользователь может расширять или сокращать. Знак "+" появляется перед пунктом, если в проекте имеется одна или более единиц, включаемых в этот пункт. Нажимая знак "+", можно раскрыть список и увидеть его содержание. Это дает возможность очень быстро и легко обращаться ко всем файлам, связанным с проектом, без необходимости обращения к другим элементам интерфейса Visual FoxPro. Несколько проектов можно открыть одновременно, и пользователи при желании могут перетаскивать файлы из одного проекта в другой. Очень удобно свернуть Project Manager к виду, похожему на панель инструментов, перемещая его к верхней или нижней части экрана. При этом не теряется возможность работы с нужными данными в нужном месте экрана, то есть реализуется так называемый "эффект отрывания закладки", как это видно из рис. 3.10.

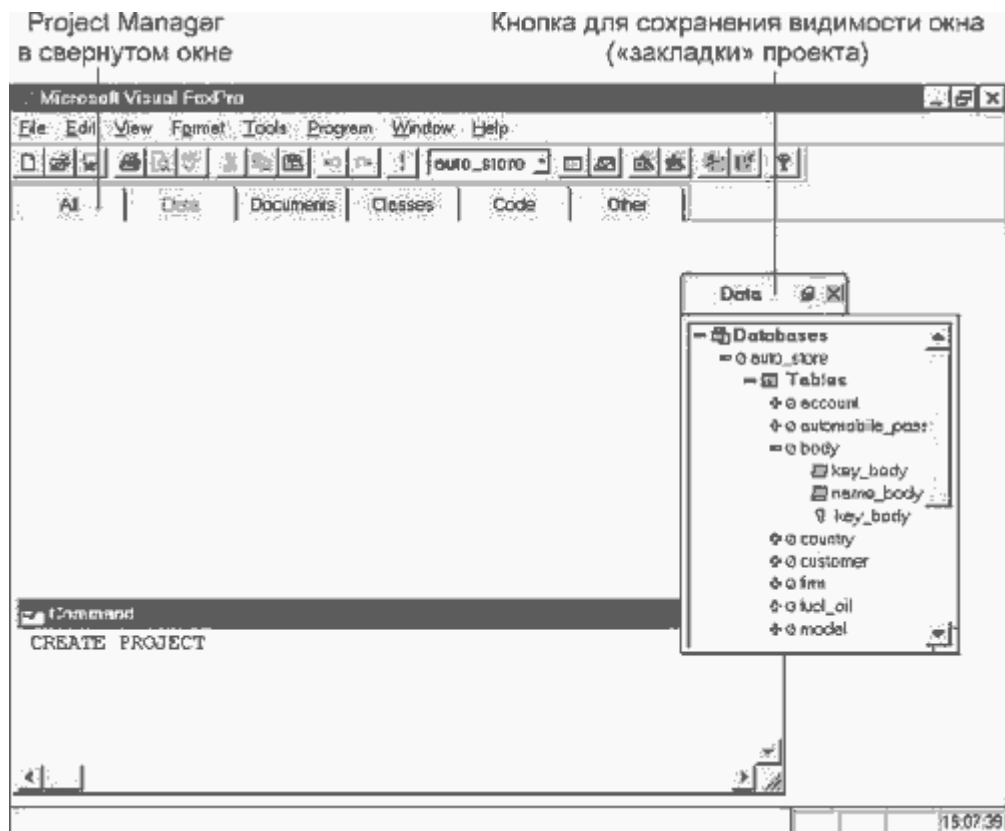


Рис. 3.10.

Database Designer

Database Designer отображает все таблицы, просмотры и связи, содержащиеся в базе данных, и позволяет визуальными средствами создать Контейнер Базы данных.

Контейнер Базы данных в Visual FoxPro 3.0 - это архив для всех связанных таблиц, локальных или внешних просмотров и соединений. Когда пользователь создает соединение с внутренними данными, оно сохраняется в Контейнере Базы данных. Если пользователь открывает Контейнер Базы данных, соединения, которые были созданы для этой базы данных, активизируются так же, как все просмотры, процедуры БД, таблицы и их связи. Когда связи установлены внутри Контейнера Базы данных, они постоянны во всей прикладной программе. Это означает, что если несколько таблиц используется для форм и отчетов, связи между таблицами создаются автоматически, основываясь на связях, созданных в Database Designer.

Form Designer

Независимо от сложности прикладных программ для автоматизации обработки информации, почти все они требуют использования экранных форм. Таким образом, Form Designer - наиболее часто используемый инструмент в разработке прикладной программы. Три панели инструментов значительно облегчают работу. С помощью панели инструментов Layout легко выравнивать объекты, панель инструментов Form Controls позволяет быстро разместить в форме элементы управления, панель инструментов Color Palette дает возможность изменения цвета элементов формы и элементов управления. Используя окно Properties с закладками, можно также осуществлять быстрый доступ к свойствам размещенного в форме объекта. Visual FoxPro 3.0 сделал простым связывание объектов и данных с помощью Data Environment Designer. Data Environment Designer визуальное представление используемых в форме таблицы и отношения между ними и по принципу работы похож на Database Designer. Сложные формы управления данными могут быть созданы путем перетаскивания полей и элементов управления на поверхность проекта формы из окна Data Environment Designer. Form Designer тесно интегрирован с объектной моделью Visual FoxPro. Например, при проектировании формы разработки могут сохранять группу объектов как класс прямо из Form Designer.

Visual Class Designer

Для эффективного использования новых методов объектно-ориентированного программирования приходится довольно часто заниматься созданием и изменением таких загадочных объектов, как классы. Новые объектно-ориентированные расширения языка в Visual FoxPro дают программистам возможность создавать классы путем написания соответствующего кода. Однако поверьте, лучше доверить эту работу Visual Class Designer. С его помощью можно быстро разработать собственный класс на основе базового класса Visual FoxPro или любого ранее разработанного класса, включенного в визуальную библиотеку. Целый ряд таких библиотек вы найдете в профессиональной версии СУБД - они имеют расширение VCX. Visual Class Designer позволяет создавать свои собственные свойства и методы. Как только свойство или метод определены, они появляются в списке окна Properties. Существенное достоинство Visual Class Designer заключается в том, что его интерфейс и методы работы сходны с Form Designer, что весьма облегчает работу с этим инструментом. Ну а если вы решили разработать не визуальный класс, устраивайтесь поудобнее за клавиатурой и начинайте пользоваться клавишей **F1**.

Query / View Designer

Запросы позволяют просматривать данные из полей одной или нескольких таблиц, отвечающих установленным критериям. Как и в предыдущих версиях FoxPro, можно создать SQL-запрос с помощью Query Designer (QDBE) и сохранить его в виде кода SQL как отдельный файл с расширением QPR. Результаты запроса могут быть выведены в окно Browse, курсор, таблицу, на график, экран, в отчет или этикетку.

View Designer имеет аналогичный интерфейс и также позволяет организовать просмотр с использованием языка SQL на основании одной или нескольких таблиц, но при этом имеется возможность вывести результаты запроса только в курсор. При желании после изменения данных в курсоре может произойти адекватное изменение данных и в исходных таблицах.

В отличие от запроса, просмотр хранится в файле БД. Просмотр является составной частью базы данных и может выполняться для локальных таблиц или внешних данных с помощью технологии ODBC.

Connection Designer

Для использования внешних данных (других форматов) или данных, расположенных на сервере, с помощью Connection Designer можно настроить соединение с требуемым источником данных посредством подключения соответствующего драйвера ODBC. Созданное соединение сохраняется как часть базы данных и содержит информацию относительно того, как обратиться к специфическому источнику данных.

Report/Label Designer

Позволяет визуально создать отчет или этикетку. В отчете может использоваться группировка данных, переменные, итоги и подытоги, заголовок, верхние и нижние колонтитулы страниц и групп данных и заключительный раздел для суммирования данных по всему отчету. При переходе в процессе печати от одной части отчета к другой могут вызываться пользовательские функции, существенно расширяющие возможности обработки различных ситуаций. Имеется режим предварительного просмотра, возможность вывода полей с плавающей длиной и вертикальной растяжкой. Работа над подготовкой структуры данных облегчена за счет использования в отчете такого вспомогательного средства, как Environment Designer.

Menu Designer

Позволяет визуально разрабатывать меню в стиле главного меню Visual FoxPro с последующей генерацией исходного кода в программный файл с расширением MPR. При этом разрабатываемое меню может быть использовано вместо или в дополнение к основному.

Вспомогательные средства разработчика

Забота о пользовательском интерфейсе - пожалуй, основная "головная боль" разработчика при написании прикладной программы. Компанией Microsoft, верной своим традициям заботы о разработчиках, в Visual FoxPro были добавлены многочисленные Мастера (Wizards), которые помогают программистам создавать таблицы, формы, отчеты и запросы, а также организовывать взаимодействие с другими прикладными программами Microsoft типа Word и Excel. В табл. 3.4 приведен список Мастеров, имеющихся в Visual FoxPro.

Таблица 3.4. Мастера в Visual FoxPro

Мастер	Описание
Table	Создание таблиц из заранее определенных наборов или отдельных полей
Query	Создание перекрестной таблицы (Cross Tab) Создание диаграмм или графиков на основе MS Graph Создание просмотров Создание запросов Создание просмотров для внешних данных
Form	Создание формы на основе данных из одной таблицы Создание формы "Один ко многим"
Report	Создание отчета с итогами и подытогами Создание отчета "Один ко многим" Создание отчета на основе данных из одной таблицы
Label	Создание почтовых этикеток или карточек
Mail Merge	Создание документа Word для рассылки по адресам
Pivot Table	Создание сводных таблиц, которые могут быть помещены в форму или MS Excel
Import	Импортирование данных в формат Visual FoxPro
FoxDoc	Документирование прикладных программ
Setup	Создание инсталляционной программы для распространения прикладных программ
Upsizing	Перемещение файлов данных FoxPro в файлы SQL Server

Все Мастера доступны из пункта меню Tools, но пользоваться многими из них можно и работая в Project Manager или из других визуальных средств, где мы выбираем команду New. В то же время, результаты нашей работы в Мастерах всегда можно "отполировать" в соответствующем Конструкторе (Designer).

Таким образом, работа с Конструкторами и Мастерами тесно увязана, и сделано это очень даже неплохо.

Пользоваться Мастерами исключительно легко. Все они имеют единый интерфейс, четко формулируют вопросы и задачи на каждой стадии работы, а развитые возможности перетаскивания объектов (Drag and Drop) делают работу с данными быстрой и приятной. Нововведение относительно версии 2.6 - на финише добавлена кнопка предварительного просмотра, которая позволяет убедиться в правильном результате и при необходимости произвести изменения с меньшей потерей времени.

При построении формы с помощью Мастера формы мы можем выбрать один из пяти стилей ее оформления:

- стандартный с выделением полей цветом;
- с выделением полей подчеркиванием;
- с выделением полей цветом и тенью (трехмерный эффект);
- размещение отдельных полей в рамках;
- рельефный.

Возможно выбрать форму с текстовыми управляющими кнопками, кнопками с пиктограммами или вообще без кнопок.

На рис. 3.11 в качестве примера приведено одно из диалоговых окон Мастера форм.

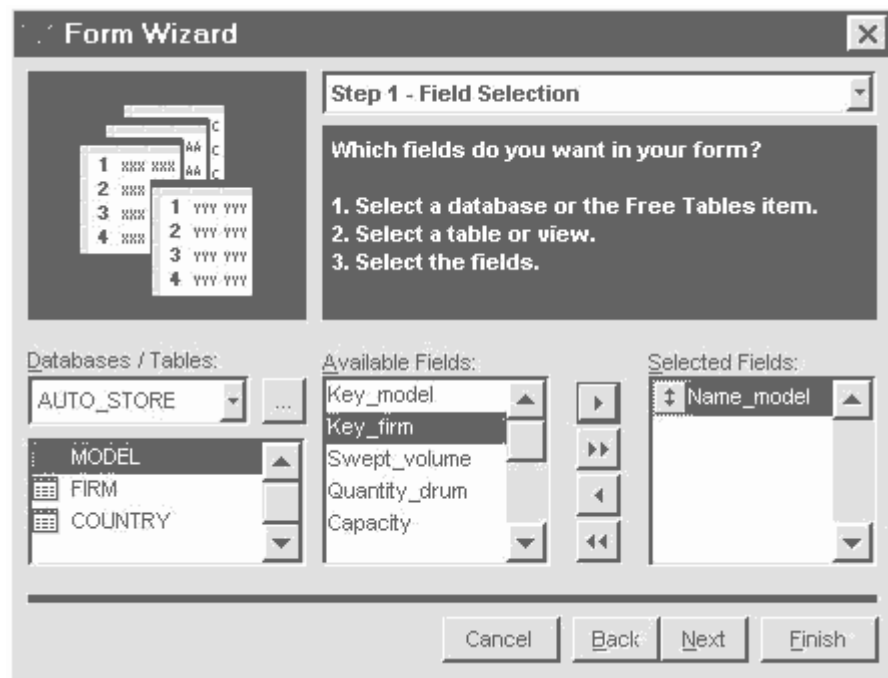


Рис. 3.11.

Мастер отчетов предлагает три стиля:

- деловой;
- бухгалтерский (с разграфленными таблицами);
- представительский.

Имеется возможность выбора числа колонок, ориентации расположения полей и ориентации расположения отчета на бумаге.

В новой версии Мастера являются более открытой для программиста структурой, чем это было раньше. Мы можем вместо "штатного" запустить свой Мастер, изменив системную переменную `_WIZARD`. Так как значительная часть элементов Мастеров задается через соответствующие регистрирующие таблицы (файлы DBF), изменяя в них данные, можно изменить и набор элементов на соответствующем этапе работы, например, стили оформления отчетов или форм.

3.3. Access

Microsoft Access - это самая популярная сегодня настольная система управления базами данных. Ее успех можно связывать с великолепной рекламной кампанией, организованной Microsoft, или включением его в богатое окружение продуктов семейства Microsoft Office. Вполне возможно, что это так. Но корень успеха скорее всего заключается в прекрасной реализации продукта, рассчитанного как на начинающего, так и квалифицированного пользователя. Не будем сейчас вдаваться в подробности сравнения отдельных характеристик Access и его основных конкурентов, например Paradox for Windows или Lotus Approach. Эта тема прекрасно освещена в периодической компьютерной печати.

СУБД Access 7.0 для работы с данными использует процессор баз данных Microsoft Jet 3.0, объекты доступа к данным и средство быстрого построения интерфейса - Конструктор форм. Для получения распечаток используются Конструкторы отчетов. Автоматизация рутинных операций может быть выполнена с помощью макрокоманд. На тот случай, когда не хватает функциональности визуальных средств, пользователи Access могут обратиться к созданию процедур и функций. При этом как в макрокомандах можно использовать вызовы функций, так и из кода процедур и функций можно выполнять макрокоманды.

Несмотря на свою ориентированность на конечного пользователя, в Access присутствует язык программирования Visual Basic for Application, который позволяет создавать массивы, свои типы данных, вызывать DLL-функции, с помощью OLE Automation контролировать работу приложений, которые могут функционировать как OLE-серверы. Вы даже можете целиком создавать базы данных с помощью кодирования, когда в этом появляется необходимость.

MS Access из всех рассматриваемых средств разработки имеет, пожалуй, самый богатый набор визуальных средств. Тем не менее кодировать в Access приходится - исходя из собственного опыта авторы берутся утверждать, что ни одно приложение, не предназначенное для себя лично,

создать хотя бы без одной строчки кода невозможно.

Для коммерческого распространения приложений, разработанных на Access, как мы уже писали, предназначен пакет **Access Developer Toolkit**, вместе с которым поставляются некоторые дополнения и несколько дополнительных объектов **ActiveX**.

Главное качество Access, которое привлекает к нему многих пользователей, - тесная интеграция с **Microsoft Office**. К примеру, скопировав в буфер графический образ таблицы, открыв **Microsoft Word** и применив вставку из буфера, мы тут же получим в документе готовую таблицу с данными из БД.

Вся работа с базой данных осуществляется через окно контейнера базы данных. Отсюда осуществляется доступ ко всем объектам, а именно: таблицам, запросам, формам, отчетам, макросам, модулям.

Посредством драйверов **ISAM** можно получить доступ к файлам таблиц некоторых других форматов: **DBASE**, **Paradox**, **Excel**, текстовым файлам, **FoxPro 2.x**, а посредством технологии **ODBC** - и к файлам многих других форматов.

Access 7.0 может выступать как в роли **OLE** контролера, так и **OLE** сервера. Это значит, что вы можете контролировать работу приложений Access из любого приложения, при условии, что оно может выступать в роли **OLE** контролера и наоборот.

Встроенный **SQL** позволяет максимально гибко работать с данными и значительно ускоряет доступ к внешним данным.

Пользователям, малознакомым с понятиями реляционных баз данных, Access дает возможность разделять свои сложные по структуре таблицы на несколько, связанных по ключевым полям.

Наша книга посвящена построению систем обработки данных. Этот процесс значительно различается на разных предприятиях и фирмах в зависимости от объема данных, которые они обрабатывают. Естественно, Access - это типичная настольная база данных. В то же время на небольшом предприятии с количеством компьютеров не больше 10, ресурсов Access вполне может хватить для обслуживания всего делопроизводства, естественно, в связке с **Microsoft Office**. То есть все пользователи могут обращаться к одной базе данных, установленной на одной рабочей станции, которая не обязательно должна быть выделенным сервером. Для того чтобы не возникали проблемы сохранности и доступа к данным, имеет смысл воспользоваться средствами защиты, которые предоставляет Access. При этом вы можете воспользоваться Мастером, если не уверены, что сами правильно установите права и ограничения для пользователей.

В отличие от других рассматриваемых средств разработки, СУБД Access имеет русифицированный интерфейс и частично переведенный на русский язык файл контекстной помощи. Как мы уже отмечали, причина этого отрадного факта заключена в позиционировании этой СУБД на конечного пользователя. Мы опустим описание интерфейса Access, так как его понимание облегчено не только русским языком, но и сохранением общего подхода, принятого в построении интерфейса всех продуктов **Microsoft** для **Windows**.

При создании многих объектов и элементов управления в Access предоставляется несколько возможностей реализации поставленной задачи. Как правило, большая часть объектов создается визуально, путем нажатия кнопки **Создать**. При этом необходимо находиться в контейнере базы данных на той вкладке, объекты которой вас интересуют. В качестве альтернативы можно воспользоваться меню **Вставка** и выбрать в нем соответствующий объект.

Очевидно, практическая работа в СУБД начинается с создания базы данных. Уже при запуске Access перед вами появляется диалоговое окно, которое предлагает создать новую базу или открыть ранее созданную, вдобавок имеется список баз данных, с которыми вы работали недавно. Если вы выберете опцию **Запуск Мастера**, то попадете в окно **Создание**. Теперь для создания базы данных можно использовать шаблоны. Для того чтобы увидеть их список, вам необходимо перейти на вкладку **Базы Данных**. Достаточно выбрать ту базу данных, которая необходима для создания вашего приложения. При этом можно добавить в базу данных только те таблицы, которые необходимы, а в таблицах выбрать нужные вам поля. После этого вы получаете базу данных с таблицами, формами ввода и вывода. Окно контейнера базы данных показано на рис. 3.12. В табл. 3.5 приведен список Мастеров, имеющихся в Access.

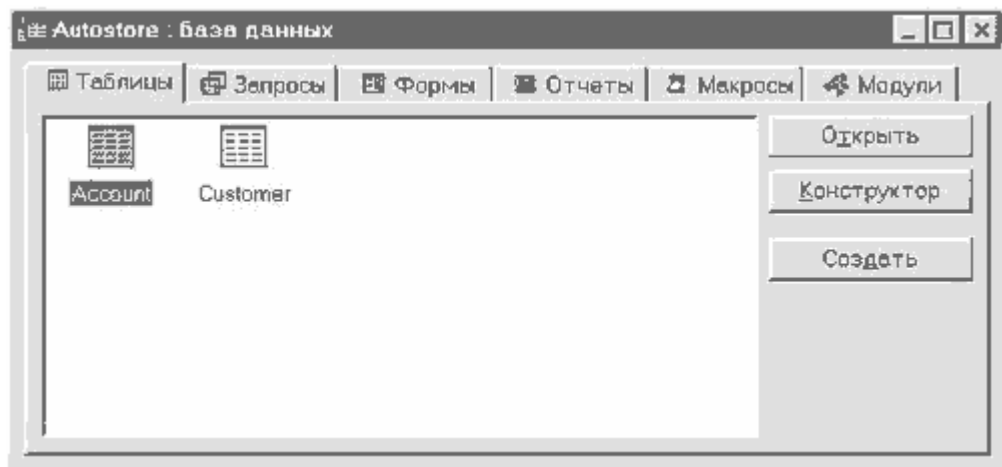


Рис. 3.12. Контейнер базы данных в Access

Таблица 3.5. Мастера в СУБД Access 7.0

Мастер баз данных	Создаются базы данных из определенного списка, возможен выбор необходимых таблиц и полей, создаются формы и отчеты.
Мастер таблиц	Создает таблицы из списка уже готовых, которые можно изменить. Интересен только на начальном этапе использования таблиц, хотя определенный круг задач можно решить, применяя только таблицы, предоставляемые мастером.
Мастер простых форм	Создает простую форму,, в которую выводятся выбранные пользователем поля из таблицы или запросы.
Мастер форм с диаграммой	Создает форму с диаграммой, отражающей данные для полей из таблиц и запросов, которые служат источником данных для формы.
Мастер форм со сводной таблицей Microsoft Excel	Создает форму, в которую включен объект "страница Excel" со сводной таблицей.
Мастер построения кнопок	Создает кнопки в форме или отчете с выбранными вами свойствами и функциональностью.
Мастер построения групп	Создает группу переключателей, которая может содержать множество кнопок, флажков, выключателей.
Мастер построения списков	Создает списки на основе полей из таблиц и запросов, SQL выражений или предопределенного набора значений.
Мастер построения комбинированных списков	Создает комбинированные списки на основе полей из таблиц и запросов, SQL выражений или заранее предопределенного набора значений.
Мастер построения подчиненных форм	Создает подчиненную форму, которая может служить аналогом объектов Grid или Browse в других системах управления данными.
Мастер создания отчета	Создает отчет, в который выводятся выбранные пользователем поля из таблицы или запросы, с возможностями установки группировки и сортировки.
Мастер создания наклеек	Позволяет создавать наклейки как стандартных, так и иных размеров.

Мастер создания отчетов с диаграммой

Позволяет выводить на печать диаграммы, внешний вид которых зависит от данных в таблице или запросе, являющихся источником данных для отчета

Дополнительно к перечисленным возможностям, все созданные формы вы можете редактировать с помощью вспомогательных диалоговых окон. При первом знакомстве с Access такой способ создания баз данных поможет больше, чем сотни страниц документации.

В процессе изучения Access необходимо как можно чаще обращаться к команде **Параметры** из меню **Сервис**. После выбора этого пункта на экран выводится диалоговое окно со множеством вкладок. В настоящий момент нас интересует вкладка с заголовком **Общие** (по-видимому, имеются в виду параметры), которая представлена на рис. 3.13. На этой странице в текстовое поле с заголовком **Рабочий каталог** внесите путь к той папке, в которой вы собираетесь хранить ваши файлы. Если при программном способе создания баз данных вам необходим каталог, отличный от текущего рабочего, то указывайте полный путь для создаваемого файла.

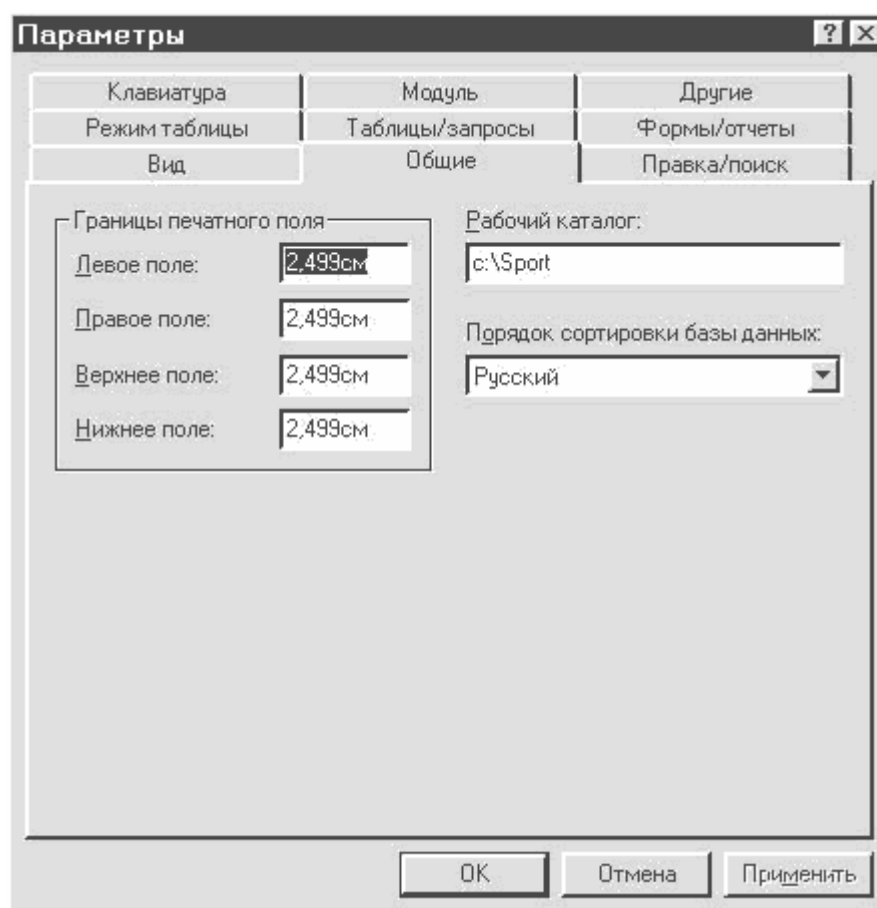


Рис. 3.13.

Первое, с чего начинается работа с базой данных - создание таблиц. После нажатия кнопки **Создать** вам будет предоставлена возможность выбора одного из пяти вариантов действий, которые приведут к появлению в вашей базе новой или присоединенной таблицы. Эти способы описаны в табл. 3.6.

Таблица 3.6. Способы создания таблиц в СУБД Access

Способ создания	Описание
Режим таблицы	Первоначально вам предоставляется таблица с тридцатью полями, куда необходимо ввести данные. После ее сохранения Access сам решает, какой тип данных присвоить каждому полю. Трудно представить человека, хоть раз в жизни создававшего таблицы и понимающего,,

что такое типы данных, который пользуется данной возможностью. Как недостаток этого способа следует отметить невозможность создать таблицу с полями примечаний.

Конструктор таблиц	После выбора этой опции открывается Конструктор таблиц, в котором вам необходимо самостоятельно создавать поля, выбирать типы данных для полей, размеры полей и, если это необходимо, устанавливать свойства полей. Подробно работа с Конструктором таблиц описана в главе 6 .
Мастер таблиц	Из predetermined набора таблиц вы можете создать таблицу по своему вкусу. Возможно, что некоторые таблицы целиком подойдут для вашего приложения, не стесняйтесь,, используйте их, все средства хороши для того, чтобы побыстрее завершить проект (рис. 3.14).
Импорт таблиц	Позволяет импортировать данные из таблиц других приложений в базу данных. Новые таблицы теряют непосредственную связь с другими приложениями. В появившемся диалоговом окне вам необходимо выбрать тип файла и имя импортируемого файла. Тип ODBC позволяет импортировать данные практически любого формата, подробнее об ODBC см. в главе 8 .
Связь с таблицами	Очень похоже на предыдущий пункт, но при этом таблица остается в своем формате, то есть может использоваться несколькими приложениями.

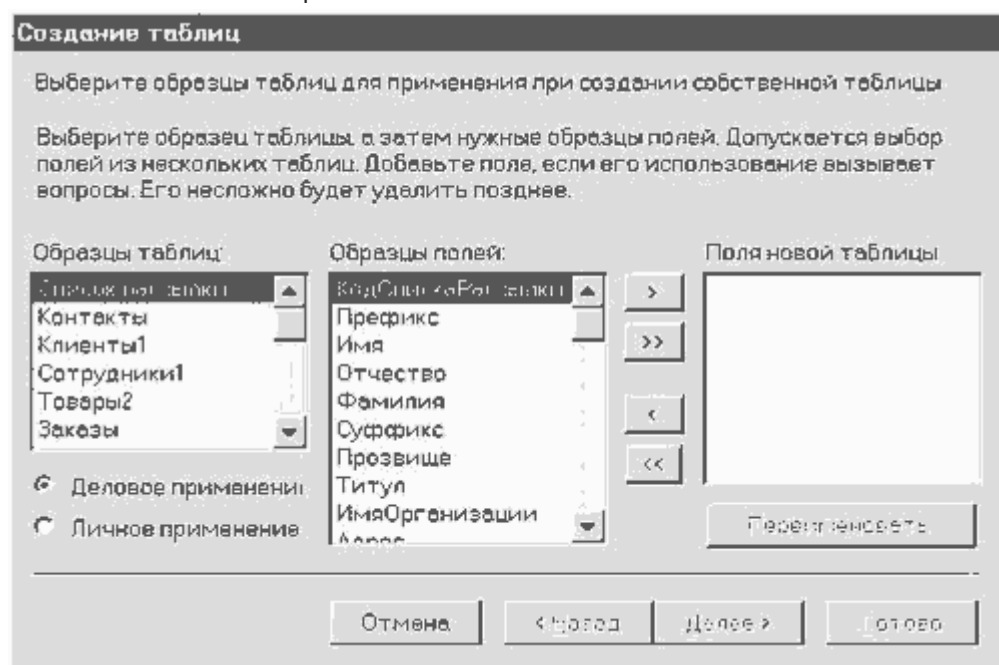


Рис. 3.14.

При работе с такими объектами, как таблица, форма, запрос, отчет, вы легко можете переключаться между режимами Конструктора, Таблицы, Формы, SQL, Предварительного просмотра. Каждому объекту соответствует присущий ему режим. Для перехода из режима в режим используйте значок, который появляется на панели инструментов, соответствующей каждому объекту (рис. 3.15), если, конечно, вы не модифицировали панели по-своему. Можно также воспользоваться меню **Вид**, которое, как практически и все остальные меню, динамически изменяется в зависимости от активного в текущий момент объекта.

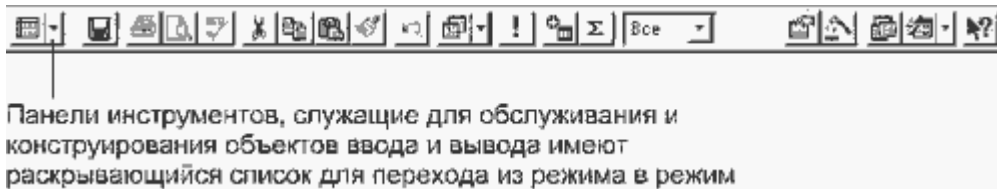


Рис. 3.15.

Конструируя таблицу, вы можете легко переходить в режим таблицы и, при обнаружении каких-то недостатков, возвращаться назад для корректировки сделанного. При этом ничто не ограничивает вас по времени модернизации. Вы можете совершить ее и сегодня, и завтра, и даже через год. Не каждая система может предоставить возможность одним движением мыши перейти из Конструктора в режим Таблицы или Формы.

Запросы

Система построения запросов в Access не имеет себе равных среди СУБД массового использования. Нетрудно поверить, что какая-нибудь третья фирма построила очень дружелюбный по интерфейсу и необыкновенно удачный продукт, который легко позволяет строить запросы самому ленивому и мало-опытному пользователю. Но кто о ней знает? Количество пользователей этого гипотетически прекрасного продукта исчерпывается первыми тысячами, возможно, в США на порядок больше. Пользователей Access и приложений, построенных на его основе, - десятки миллионов. Практически все типы запросов, которые можно построить программно, в Access можно создать визуально. Исключение составляют сквозные запросы (SQL pass-through), запросы на изменение структуры данных (DDL) и запросы объединения.

В Access вам предоставляется возможность создавать самые разнообразные запросы выборки, причем подчеркиваем, что они модифицируют исходные данные. А ведь эта возможность реализована далеко не в каждом пакете. Можно сказать, что Access был одним из пионеров практического применения запросов, изменяющих данные в таблицах, на основе которых они построены. Именно здесь кроются резервы ускорения работы с данными. Это очень важно, так как у оппонентов Access любимая тема для разговоров - обсуждение резкого замедления скорости работы с данными у Access при увеличении размеров таблиц.

Мы уже не будем говорить про развитую систему фильтров. Microsoft слишком занят глобальными проблемами завоевания Internet, и ему некогда обращать внимание оппонентов (и просто заклятых врагов) на такую мелочь. О том, насколько ускоряется работа при наличии индексов, предоставляем вам сделать выводы самостоятельно сразу же после прочтения этой книги и проведения соответствующих экспериментов.

Также визуально вы можете построить запросы добавления, удаления, обновления, создания таблиц. Причем таблицу можно создать в другой базе данных. Обратим ваше внимание на перекрестный запрос. Пять-десять минут, которые вы потратите на его освоение, в дальнейшем сэкономят недели работы.

Сквозные запросы делают вас поистине всесильным разработчиком. Вы можете контролировать работу любого сервера баз данных, находясь в любимой среде Access. Правда, предварительно необходимо запросить у Администратора сети те же права, которые имеются у него.

Запросы изменения структуры данных, позволяющие вам создавать новые таблицы и индексы, а также изменять структуру существующих таблиц, авторы нашли очень полезными при работе с базами данных Access из других приложений, например из Visual FoxPro или Excel, используя сквозные запросы оттуда. Другая причина, которая может побудить вас использовать их, - нежелание разбираться с синтаксисом объектов DAO, так как они обладают такой же функциональностью.

Для построения запросов воспользуйтесь одним из уже известных вам способов создания новых объектов. В появившемся диалоге Новый запрос будет предложено выбрать из способов построения запросов. Правда, последние три Мастера строят запросы со специфическими характеристиками, необходимыми для особо сложных случаев, которые, возможно, не понадобятся в первый день знакомства с Access, но наверняка пригодятся в дальнейшем. Если вы мало знакомы с построением запросов, то вначале попробуйте использовать Мастер, который называется Простой запрос. Затем откройте этот же запрос в режиме Конструктора. Как мы уже говорили, вы легко можете переходить из одного режима в другой. Для запросов доступны три режима: Конструктор, показанный на рис. 3.16, SQL - на рис. 3.17 и Таблица - на рис. 3.18. Режим Конструктора и SQL взаимосвязаны, любые изменения в одном из режимов приводят к изменениям в другом. Опять же следует отметить, что не каждая среда создания баз данных может предоставить вам такую возможность. Обычно вам любезно предоставляют возможность посмотреть код полученного SQL-запроса в режиме только для чтения. Настоятельно советуем

как можно чаще переключаться в режим SQL, что поможет вам быстрее освоить синтаксис SQL. Чем лучше вы освоите SQL, тем легче будет в дальнейшем. Как ни хороши были бы визуальные средства построения запросов, иногда легче набрать несколько операторов вручную. При переходе в режим Таблицы вы увидите результаты вашего запроса. Microsoft постарался, чтобы вы не только пользовались запросами, но и активно осваивали их.

Как указывалось выше, фильтры - это одна из наиболее сильных сторон Access. Фильтры строятся с помощью запросов или установкой критериев. Но очень сложно построить запросы на все случаи жизни. Для того чтобы хоть как-то облегчить эту задачу, используются параметрические запросы. Создав параметрический запрос, вы даете возможность пользователю вводить значения для отбора данных - перед ним после запуска запроса обязательно возникнет диалоговое окно с просьбой указать критерии отбора, как это показано на рис. 3.19.

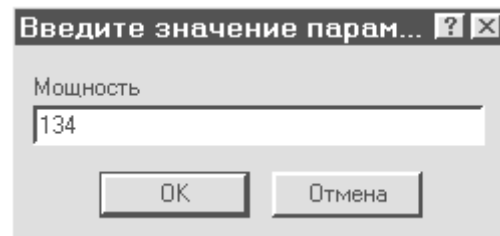


Рис. 3.19.

Когда вы создаете запрос с помощью Конструктора или окна SQL, то загружается меню **Запрос**, которое имеет команду, так и называющуюся - **Запрос**. Именно здесь можно выбрать тип запроса, который необходимо создать.

Запросы можно составлять программным путем. При этом различаются два подхода. Первый - это запуск непосредственно команд SQL. Для этого необходимо создать переменную строкового типа и запустить ее с помощью макрокоманды RunSQL. Второй способ - это использование объектов доступа к данным.

Формы

После создания таблиц и запросов можно организовать ввод данных с помощью формы. Иногда пользователи предпочитают работать с данными, выведенными в виде таблицы, которая напоминает им одну из рабочих книг. Но часто бывает удобнее организовать данные в виде формы ввода, когда видно только одну запись, поля которой расположены в нужном порядке. При этом вы можете в полной мере проявить здесь свои способности к дизайнерскому искусству, если таковые имеются.

При желании создать простейшую форму воспользуйтесь опцией автоформа. Это самый простой способ создания формы. Вы получаете форму с набором текстовых полей, перед которыми выводятся названия или заголовки полей, если последние имеются для поля, как это видно на рис. 3.20. В дальнейшем, по мере освоения пакета, вы сможете использовать эту возможность для создания основы вашей формы с целью дальнейшего ее усовершенствования.

Для создания автоформы воспользуйтесь раскрывающимся списком Новые объекты на панели инструментов, которая показана на рис. 3.21, либо используйте команду меню **Вставка**. Если вы решите создавать новую форму, воспользовавшись кнопкой Создать на вкладке Формы, то в списке доступных режимов создания будут присутствовать автоформы, причем трех типов: состоящая из колонок (ее вы можете построить предыдущим способом), ленточная и табличная. Последняя совпадает с обычным табличным режимом вывода данных для таблицы, но это все-таки форма, потому что вы легко можете устанавливать выводимые поля, что очень трудно сделать для таблиц.

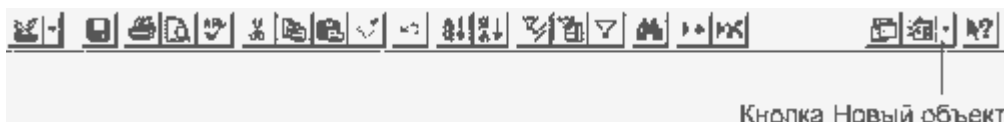


Рис. 3.21.

После того как вы решите создать форму, перед вами появится диалоговое окно Новая форма, показанное на рис. 3.22, в котором имеется целых семь опций. Обращаем ваше внимание, что надо выбрать источник данных, в противном случае для вывода данных надо применять более сложные технологии. Можете выбрать любой способ, но следует отметить, что на самом деле это не семь способов создания форм.

К примеру, опция Простая форма, которая запускает Мастер создания форм, по своим

возможностям раскладывается на три следующие за ними автоформы, но с предоставлением возможности выбрать из исходных таблиц нужные вам поля, задать свой заголовок, выбрать один из стилей оформления.

Основные этапы построения формы с помощью Мастера показаны на рис. 3.23- 3.27.

Рис. 3.23.

Рис. 3.24.

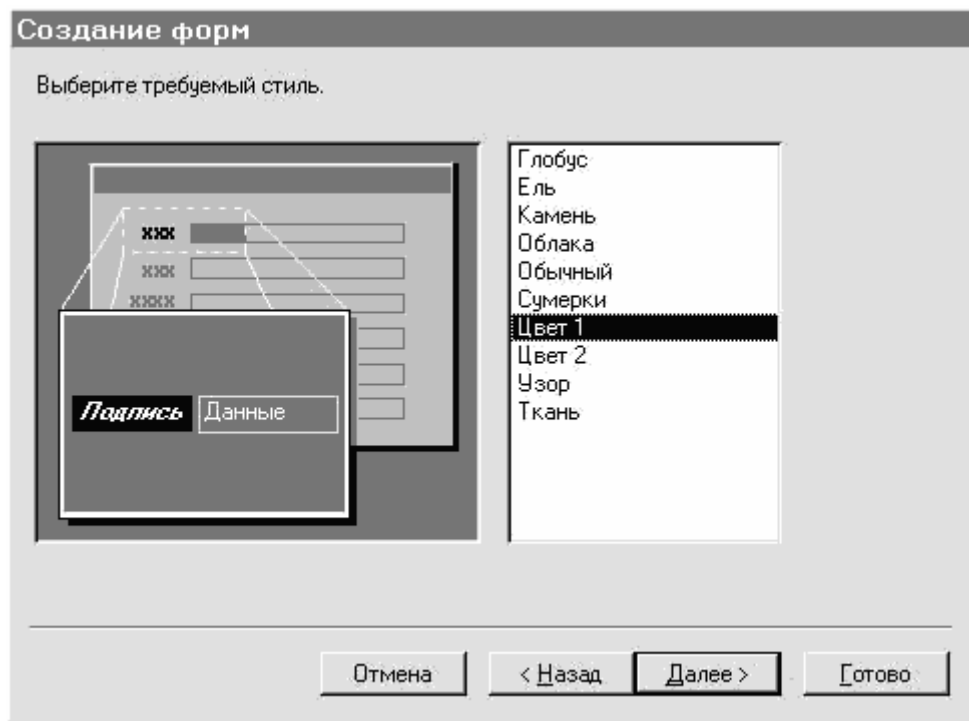


Рис. 3.25.



Рис. 3.26. Выбор названия формы и дальнейших действий по ее использованию

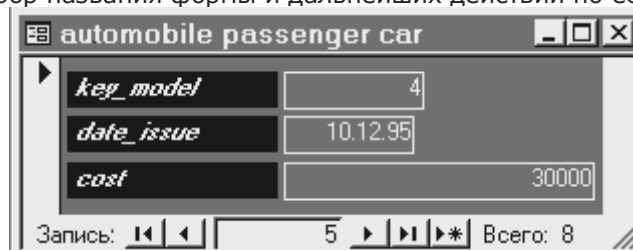


Рис. 3.27. Готовая форма

Последние опции - Сводная таблица и Диаграмма - позволяют создавать достаточно специализированные по своим задачам формы и активно используют OLE технологии. Новичкам рекомендуется начинать с автоформ, потом, осознав различия между различными видами форм и желая применить более высокие требования к выводу данных, попробовать опцию Простая

форма. Иногда бывает полезно представить данные, организованные с помощью опций Диаграмма и Сводная таблица.

Формы, которые удовлетворяют любому, даже самому требовательному вкусу, можно создать с помощью Конструктора форм. Эффективным способом работы является быстрый выбор полей с помощью Мастера создания форм, стиля форм и дальнейшее совершенствование форм с помощью Конструктора.

Создание форм отнимает больше половины времени, затрачиваемого на создание приложений. Но именно на этом этапе работы вы можете воспользоваться наибольшим количеством средств автоматизации. Вам предоставляется большое количество встроенных объектов. Со многими объектами связаны Построители, причем число их разновидностей так велико, что позволяет построить автоматизировано 90%, а в некоторых случаях и целиком приложение. К примеру, с помощью Построителя командных кнопок вы можете создать 28 различных кнопок, функции которых простираются от перехода по записям до набора телефонного номера (рис. 3.28).

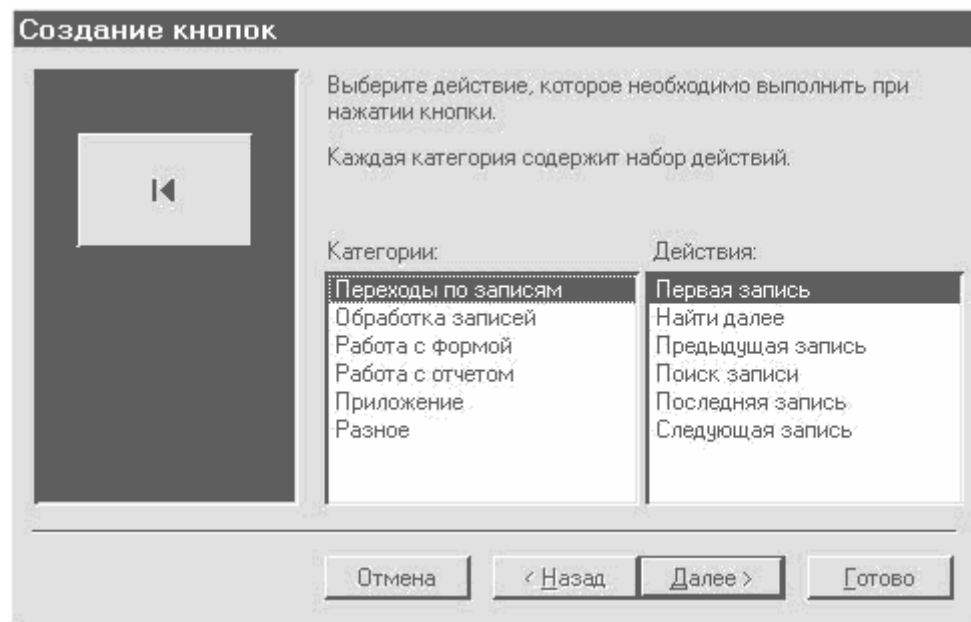


Рис. 3.28.

Помимо кнопок Мастера имеются для Групп, причем вы сами можете решить, какие объекты поместить внутрь контейнера группы, для списков, комбинированных списков, подчиненных форм, которые можно считать в Access аналогом объекта Grid, используемым в Visual FoxPro.

Каждый объект имеет большой набор свойств и событий. Событию можно присвоить макрокоманду или процедуру, которые будут вызываться при его наступлении. С помощью этого можно добиться значительной гибкости работы с формой.

Для форм доступны три режима работы: Конструктор, Форма и Таблица. Режим вывода данных имеет три вида: ленточная форма, простая форма и таблица. При работе с простой формой одновременно вы можете видеть данные только из одной записи, при ленточной - одну и более, в зависимости от того, сколько можно уместить их на экран.

В Access нет объекта типа страницы с закладками, но есть объект Разделитель страниц и метод для перехода со страницы на страницу, поэтому не трудно организовать многостраничную форму, правда, эта функциональность может быть достигнута только после написания нескольких процедур.

При работе с формой загружается своя система меню, в режиме Конструктора - одна, а в режиме формы - другая. То же относится и к панели инструментов. В режиме формы вы можете указать, какое меню и панель инструментов должны загружаться, при этом можно указывать и созданные вами.

Используя установки, которые вам доступны после выбора команды **Параметры** из меню **Сервис**, вы можете задать шаблон формы, в качестве которого может использоваться любая заранее созданная форма. Все новые формы будут создаваться на основе этой формы со всеми включенными в нее элементами управления и свойствами.

Формы и элементы управления можно создавать и модифицировать программно, но занимает это намного больше времени.

Отчеты

Отчеты наряду с формами создают представление о вашем приложении, и создание их обычно требует кропотливого труда. Компания Microsoft постаралась ваш труд облегчить. Для каждой таблицы вы можете создать *Автоотчет*. При этом Автоотчет, доступный для создания с помощью меню или кнопки Новый объект на панели инструментов База данных, создает отчет, данные в котором будут выведены в столбец. Еще один Автоотчет станет доступным при выборе кнопки Создать на вкладке Отчеты. Это ленточный автоотчет, когда данные из всех полей будут выводиться в колонку. Если вы хотите выбрать поля для отчета, а не вывести все, имеющиеся в таблице или запросе, то воспользуйтесь *Мастером отчетов*. Мастер отчетов позволяет, помимо выбора полей для отчета, сгруппировать данные по какому-нибудь полю, при этом вы можете установить интервал группировки, установить порядок сортировки, выбрать макет отчета и его стиль.

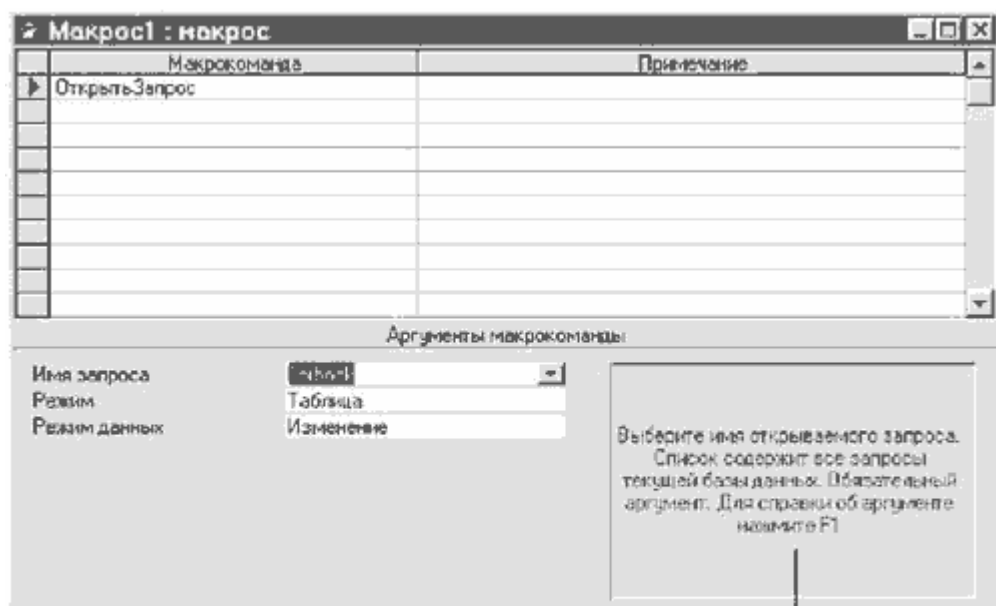
Среди прочих мастеров отметим Мастер по созданию отчета с диаграммой и построитель почтовых наклеек. Построитель почтовых наклеек - это мощное средство с разнообразными свойствами, требующее подробного изучения. Построитель почтовых наклеек по своему классу скорее можно отнести к средствам визуального проектирования.

Для построения сложных отчетов предназначен *Конструктор отчетов*. При его запуске вместе с ним загружается панель инструментов с элементами управления, которые можно размещать в различных областях проектируемого отчета путем буксировки с помощью мыши. Перед печатью отчета его можно просмотреть в окне предварительного просмотра.

Как и многие другие объекты базы данных, отчеты можно создавать программно. Обычно этот метод используется для создания собственных Мастеров.

Макросы

Макрокоманды, которые можно объединять в макросы, совершают разнообразные действия, выполнимые в СУБД Access, а с помощью параметров этим действиям можно придать гибкость, которой иначе можно добиться только путем кропотливого программирования. В Access имеется около пятидесяти макрокоманд. Для того чтобы создать макрос, вам обязательно нужно загрузить окно *Конструктора макросов*, которое состоит из двух частей (рис. 3.29). Верхняя служит для ввода макрокоманд и представляет собой таблицу, имеющую от двух до четырех колонок, в зависимости от установленных параметров. В нижней части окна Конструктора макросов вводятся аргументы макроса. В зависимости от выбранной макрокоманды число аргументов динамически меняется. Напротив каждого аргумента имеется его описание. Справа отображается пояснение для рода и типа требуемого аргумента. Очень часто аргумент можно выбрать из раскрывающегося списка.



Описание назначения
и вида аргумента

Рис. 3.29.

Для построения более сложных макросов, требующих выполнения определенных условий для запуска отдельных макрокоманд, необходимо добавить колонку Условия. После этого

макрокоманды, находящиеся в одной строке с условием, будут выполняться, только если выражение условия будет истинным. Для того чтобы распространить действие условия на последующие строки, достаточно ввести многоточие (...) в колонке условия напротив соответствующих макрокоманд.

С помощью Конструктора макросов можно создавать и меню, но все-таки более удобным средством является непосредственно Конструктор меню.

Система защиты

Access обладает лучшей встроенной системой защиты среди всех настольных приложений СУБД. Вы можете создавать группы, пользователей, присваивать права доступа ко всем объектам, в том числе и модулям. Кстати, это решает вопрос закрытия ваших процедур и функций от чужих глаз. Так как для Access нет компилятора, то необходимость защиты становится очень актуальной для разработчиков. Система защиты доступна только при открытой базе данных. Каждому пользователю можно предоставить индивидуальный пароль. Система защиты доступна как с помощью визуальных средств, так и программным путем. Если вы хотите защитить вашу базу данных даже от пользователя с именем **Admin**, то пользуйтесь услугами надстройки **Security**, которая поставляется вместе с **Access Developer Toolkit**. Помимо этого вы можете закрыть вашу базу данных от просмотра внешними программами.

3.4. Visual Basic

Visual Basic является универсальным средством программирования, но, исходя из задачи данной книги, мы будем рассматривать его возможности только с точки зрения создания приложений по обработке данных.

В отличие от большинства пакетов программ, Visual Basic не имеет главного окна, объединяющего все остальные элементы интерфейса разработчика. Каждый элемент Visual Basic имеет свое независимое окно, которое может быть убрано или расположено независимо от других в любом месте экрана, как это показано на рис. 3.30.

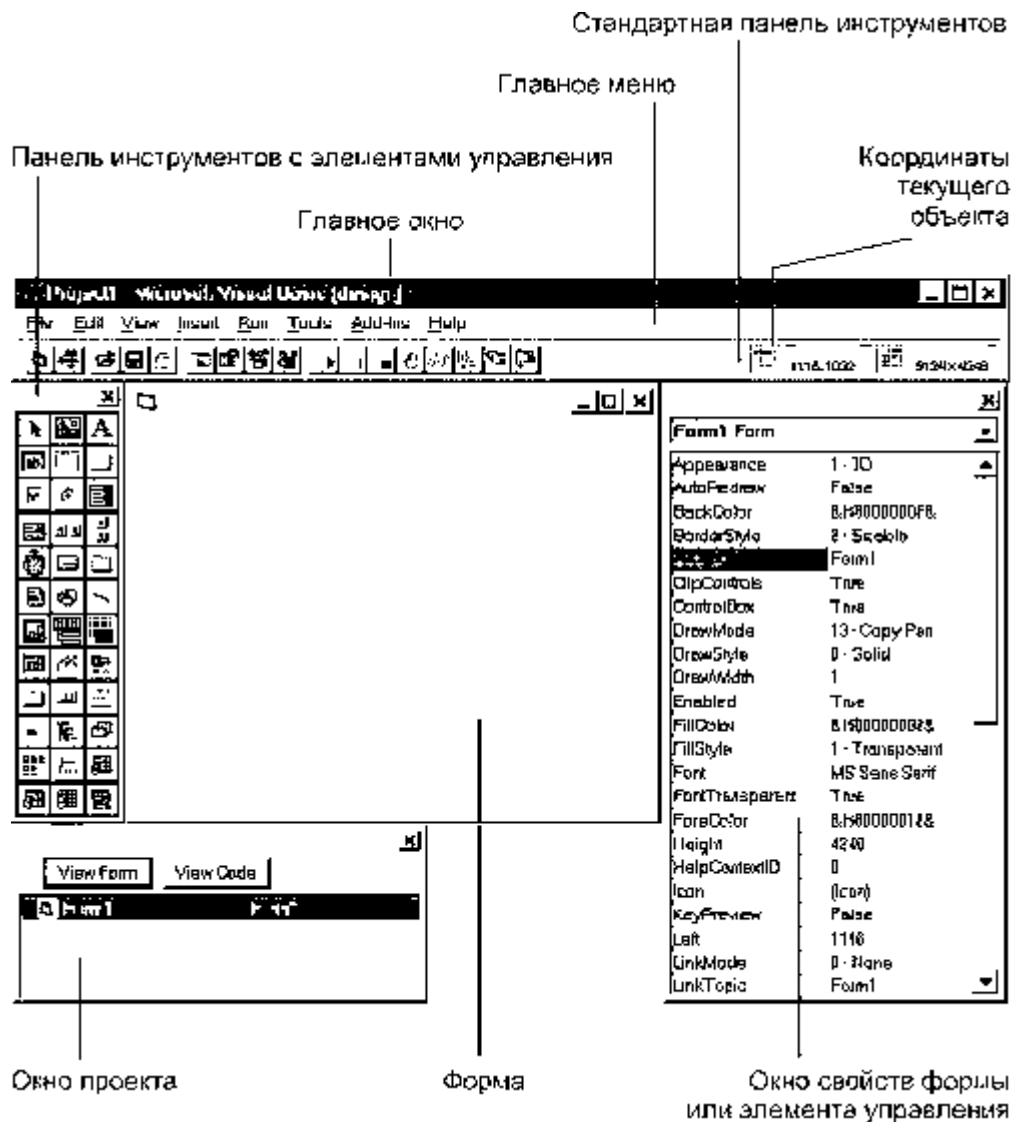


Рис. 3.30. Интерфейс Visual Basic 4.0

При запуске Visual Basic вы сразу попадаете в режим разработки и по умолчанию открывается новый проект с готовой к созданию формой. Опишем кратко назначение команд в главном меню Visual Basic. Меню File является достаточно стандартным для всех средств разработки Microsoft и содержит команды, предназначенные для работы с файлами:

- **New Project** - создание нового проекта. В Visual Basic 4.0 одновременно можно работать только с одним проектом. Если вы очень хотите работать сразу с несколькими проектами - запустите еще одну копию Visual Basic.
- **Open Project** - открытие существующего проекта.
- **Save File** - сохранение файла формы или программного модуля.
- **Save File As** - сохранение файла формы или программного модуля с другим именем.
- **Save Project** - сохранение файлов проекта.
- **Save Project As** - сохранение файлов проекта с другим именем.
- **Add File** - добавление в проект уже существующего файла формы, программного модуля, класса или ресурса.
- **Remove File** - удаление из проекта уже существующего файла формы, программного модуля, класса или ресурса.
- **Print Setup** - вызов диалогового окна установки принтера.
- **Print** - печать файлов проекта.
- **Make EXE File** - создание (компиляция) из файлов проекта исполняемого модуля в виде файла EXE.
- **Make OLE DLL File** - создание (компиляция) из файлов проекта динамической библиотеки Windows в виде файла DLL.

Меню **Edit** содержит стандартные команды, позволяющие редактировать содержимое формы и текст программы, а также выравнивать в форме элементы управления.

Меню **View** предназначено для быстрого перехода или вывода на экран различных окон программной оболочки. Две команды этого меню позволяют более эффективно работать с большими программами:

- **Procedure Definition** - позволяет быстро переместиться к коду вызываемой процедуры, если курсор находится на ее имени.
- **Last Position** - обеспечивает перемещение курсора в точку последнего изменения программного кода. Поддерживается до четырех "возвратов". Меню **Insert** предназначено для вставки в проект новых компонентов (процедуры, формы, модуля класса и т. п.). Меню **Run** служит для выполнения программы:
- **Start** - запускает программу на выполнение.
- **Start With Full Compile** - запускает программу на выполнение с предварительной перекомпиляцией всех файлов проекта.
- **End** - прерывает выполнение программы в случае неверного результата.
- **Restart** - возобновляет прерванную работу программы.
- **Step Into** - показывает выполнение последовательно всех строк кода.
- **Step Over** - показывает выполнение кода только данной процедуры.
- **Step To Cursor** - выполняет без вывода на экран часть кода до строки, на которой находится курсор.
- **Toggle Breakpoint** - устанавливает или снимает точку, в которой выполнение программы прерывается.
- **Clear All Breakpoints** - снимает все точки прерывания работы программы.
- **Set Next Statement** - устанавливает строку в программном коде, которая будет выполняться следующей.
- **Show Next Statement** - устанавливает курсор на строку, которая будет выполняться следующей.

Меню **Tools** включает доступ к дополнительным средствам разработчика:

- **Add Watch** - позволяет показать значение переменной в окне отладчика во время выполнения программы.
- **Edit Watch** - позволяет редактировать значение переменной во время выполнения программы.
- **Instant Watch** - позволяет вывести значение переменной, которое она имеет в данный момент.
- **Calls** - представляет список всех незавершенных процедур.
- **Menu Editor** - вызывает утилиту построения пользовательского меню.
- **Custom Controls** - подключает к проекту дополнительные компоненты VBX и ActiveX.
- **References** - позволяет установить ссылку на внешние объекты.
- **Get** - копирует версию проекта из программы контроля версий и управления групповой разработкой проекта (MS Visual SourceSafe 4.0) в рабочую папку с атрибутом только для чтения.
- **Check Out** - копирует выделенные файлы проекта из программы контроля версий в рабочую папку с атрибутом чтение/запись.
- **Check In** - обновляет версию проекта в программе контроля версий.
- **Undo Check Out** - прерывает команду **Check Out**, отменяя все сделанные изменения.
- **Options** - выводит на экран диалоговое окно настройки параметров среды разработки.

Меню **Add-In** служит для расширения среды разработчика. Стандартно в него входят команды для вызова:

- **Data Manager** - утилита для интерактивной работы с данными.
- **Add-In Manager** - включение дополнительных средств разработчика.

Меню **Help** предназначено для вызова контекстной помощи и справочной информации о Visual Basic.

В приложении Visual Basic могут присутствовать файлы, типы которых приведены в табл. 3.7.

Таблица 3.7. Типы основных файлов в Visual Basic

Тип файла	Расширение файла
Пользовательское приложение, включающее в себя все необходимые компоненты	EXE
База данных	MDB
Отчет	RPT
Готовые компоненты и элементы управления OLE (ActiveX)	OCX (VBX)
Проект	VBP
Форма	FRM
Двоичный код для формы	FRX
Программа	BAS
Файл ресурсов	RES
Модуль классов	CLS

Основные возможности **Visual Basic**, применимые в разработке приложений для обработки информации, могут быть реализованы благодаря наличию в нем объектов для доступа к данным - **Data Access Object (DAO)**, 32-разрядного процессора данных - **JET 3.0** и предназначенных специально для работы с данными элементов управления.

Процессор данных в **Visual Basic** поддерживает все стандартные операции по созданию, изменению и удалению таблиц, индексов и запросов. Формат БД процессора данных **Visual Basic** соответствует формату **Access**. **JET 3.0** также обеспечивает поддержку целостности и проверку вводимых и изменяемых данных на уровне полей и записей. Для изменения данных **JET 3.0** позволяет использовать язык **SQL**, который, правда, не соответствует на 100 процентов стандарту **ANSI**.

Управление базой данных обеспечивается процессором данных с помощью объектов для доступа к данным. Эти объекты позволяют разработчику программным путем, с помощью соответствующих свойств и методов **DAO**, как манипулировать данными, так и управлять структурой БД, включая ее создание. По сравнению с предыдущей версией **Visual Basic** возможности объектов для доступа к данным теперь существенно расширены. Вы можете использовать для работы с данными несколько рабочих областей, поддерживать целостность данных, включая каскадное удаление и обновление, и обеспечивать их защиту от несанкционированного доступа. Существенно сократить программный код позволяет использование коллекций.

Уникальным свойством **JET 3.0** является возможность создания копий данных (репликации БД). Для создания копии БД разработчику достаточно воспользоваться методом **MakeReplica**. При задании метода **Synchronize** выполняется согласование данных в обновляемой и оригинальной БД. Причем эти операции могут выполняться как с файлами формата БД процессора данных, так и с БД других форматов, поддерживаемых через **ODBC**.

Нельзя не отметить, что **JET 3.0** использует индексы новой, более компактной структуры, позволяющие уменьшить время их создания и ускорить процесс поиска данных.

В **Visual Basic Enterprise Edition** включены объекты для доступа к внешним данным - **Remote Data Object (RDO)** и соответствующие элементы управления - **Remote Data Control (RDC)**. Это позволяет, не прибегая к помощи процессора данных **JET 3.0**, использовать все возможности работы с курсорами на сервере, достигая максимально возможной скорости доступа к данным и минимизируя сетевой трафик.

3.5. MS SQL Server

Microsoft SQL Server 6.5 - одна из наиболее мощных СУБД архитектуры клиент-сервер. Эта СУБД позволяет удовлетворять такие требования, предъявляемые к системам распределенной обработки данных, как тиражирование данных, параллельная обработка, поддержка больших баз данных на относительно недорогих аппаратных платформах при сохранении простоты управления и использования.

MS SQL Server представляет собой систему, конечно, плохо сравнимую с рассмотренными выше СУБД. Он не предназначен непосредственно для разработки пользовательских приложений, а выполняет функции управления базой данных. Для пользовательского приложения **SQL Server** является мощным источником генерации и управления нужными данными.

Сервер имеет средства удаленного администрирования и управления операциями, организованные на базе объектно-ориентированной распределенной среды управления. **Microsoft**

SQL Server 6.5 входит в состав семейства Microsoft BackOffice, объединяющего пять серверных приложений, разработанных для совместного функционирования в качестве интегрированной системы.

Microsoft SQL Server 6.5 предназначен исключительно для поддержки систем, работающих в среде клиент-сервер. Он поддерживает широкий спектр средств разработки и максимально прост в интеграции с приложениями, работающими на ПК.

Построенная на основе технологических решений, появившихся в Microsoft SQL Server 6.0, версия 6.5 демонстрирует много значительных нововведений. SQL Server 6.5 превосходит предыдущую версию с точки зрения использования многопоточной параллельной архитектуры операционной системы для повышения производительности и масштабируемости, то есть очень эффективно использует возможность ускорения работы в том случае, если на компьютере установлено несколько процессоров.

SQL Server 6.5 имеет новую масштабируемую архитектуру блокировок, называемую Динамической блокировкой (Dinamic Locking), которая комбинирует блокировку на уровне страницы и записи для достижения максимальной производительности и подключения максимального числа пользователей.

SQL Server может тиражировать информацию в БД иных форматов, включая Oracle, IBM DB2, Sybase, Microsoft Access и другие СУБД (при наличии ODBC драйвера, отвечающего определенным требованиям).

Хранимые процедуры, поддерживающие OLE Automation, позволяют разработчику применять практически любой инструмент из тех, что поддерживают OLE, в целях создания хранимых процедур для SQL Server. Visual Basic 4.0 поддерживается посредством новой 32-разрядной DB-Library (OCX). Многочисленные расширения языка Transact-SQL включают расширенную поддержку курсоров, возможность использования команд определения данных внутри транзакций и т. д.

Microsoft SQL Server 6.5 содержит Ассистент администратора. Этот инструмент позволяет назначать основные процедуры сопровождения базы данных и определять для них график выполнения. Операции по сопровождению баз данных включают проверку распределения страниц, целостности указателей в таблицах (включая системные) и индексах, обновление информации, необходимой оптимизатору, реорганизацию страниц в таблицах и индексах, создание страховочных копий таблиц и журналов транзакций. Все эти операции могут быть установлены для автоматического выполнения по заданному администратором графику.

Пакет Enterprise Manager включает утилиту, позволяющую переносить некоторые или все объекты из одной базы данных в другую. Используя эту утилиту, разработчик или администратор может:

- выполнять копирование объектов любого типа с указанием, какого типа объекты подлежат копированию (или копировать все объекты всех типов);
- переносить схему базы данных вместе с данными или без них;
- дополнять или замещать существующие данные;
- уничтожать объекты в базе-приемнике перед копированием схемы;
- для копируемого объекта включать объекты, от него зависящие;
- использовать стандартные настройки генерации кода создания/удаления объектов или использовать собственные;
- определять момент выполнения переноса объектов: немедленно, однократно в определенный момент времени, многократно по определенному графику.

Сервер, который получает объекты, должен быть Microsoft SQL Server версии 6.5. Сервер-источник может быть Microsoft SQL Server версии 4.x или 6.x или сервер Sybase.

SQL Server предоставляет возможность создания страховочных копий и восстановления индивидуальных таблиц. Загрузка таблицы может быть выполнена либо из копии индивидуальной таблицы, либо из копии базы данных. Загрузка индивидуальных таблиц может оказаться хорошим решением при необходимости восстановления данных после сбоя, когда загрузка всей базы данных неэффективна. Тем не менее создание страховочных копий всей базы данных и журнала транзакций остаются основой стратегии резервного копирования.

Для эффективной работы с данными SQL Server имеет целый набор специальных инструментов (рис. 3.31). Характеристика основных из них приведена в табл. 3.7.

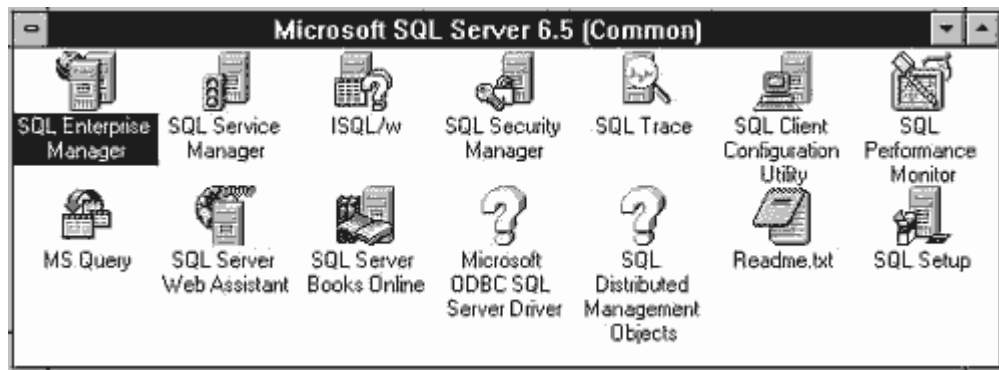


Рис. 3.31.

Таблица 3.7. Характеристика основного инструментария SQL Server 6.5

Графический инструмент	Описание
SQL Setup	Используется для установки нового, модификации установленного программного обеспечения и удаления SQL Server с диска. Программа Setup также может быть использована для изменения опций сетевой поддержки, подключения языка, перестройки БД Master и установки опций доступа к данным.
SQL Service Manager	Используется для старта и остановки служб SQL Server (SQL Server и SQL Executive).
ISQL/w	Позволяет вводить выражения и хранимые процедуры Transact-SQL в графическом интерфейсе запросов.
SQL Security Manager	Позволяет управлять бюджетами пользователей серверов SQL.
SQL Enterprise Manager	Обеспечивает управление с сервера или с рабочей станции. Он позволяет вам выполнять задачу системного администрирования, используя удобный графический интерфейс.
SQL Client Configuration Utility	Устанавливает информацию соединения сервера для клиентов.
SQL Transfer Manager	Обеспечивает легкий графический способ переноса объектов и данных с одного сервера на другой.
SQL Trace	Графическая утилита, позволяющая администраторам и разработчикам отслеживать и фиксировать активность клиентских приложений, обращающихся к Microsoft SQL Server 6.5. SQL Trace может в реальном времени отображать все аспекты обращений к серверу или использовать фильтры, отображающие информацию о действиях конкретных пользователей, приложений или машин.

3.6. Руководство для покупателя

Жизнь не стоит на месте и сейчас трудно себе представить, что каких-то десять лет назад в существовавшем еще тогда СССР невозможно было купить легальную копию такой популярной СУБД, как FoxPro. Сей факт, конечно, способствовал выработке стойкого отсутствия интереса программиста к вопросу: "Что такое легальная копия?" Но возрастающая сложность программного обеспечения и разрабатываемых с его помощью систем заставляет нас, несмотря на массу различных проблем (связанных, в основном, с необходимостью заплатить), решительно повернуться в сторону легального пакета программ. Этот вариант гарантирует получение

программ из надежного источника, в полном комплекте и с обширной документацией. Как ни странно может показаться на первый взгляд, наиболее популярный вопрос потенциального покупателя: "А что, собственно я могу купить?" Давайте посвятим ответу на этот вопрос несколько страниц нашей книги.

Средства разработки, о которых идет речь в данной книге, поставляются для розничной продажи в нескольких вариантах комплектации. Начнем с Visual FoxPro. В соответствии с общей политикой фирмы-производителя пакета Visual FoxPro - Корпорации Microsoft, программное обеспечение для разработчиков поставляется в двух вариантах:

- Стандартная версия предназначена для хорошо подготовленного пользователя, который хочет найти наиболее оптимальный вариант работы с большими объемами данных, программистов или студентов, разрабатывающих программное обеспечение не в коммерческих целях.
- Профессиональная версия предназначена для программистов, разрабатывающих коммерческое прикладное программное обеспечение. В связи с этим версия включает средства создания независимо работающего пользовательского приложения (исполняемого модуля). Профессиональная версия включает также максимальное количество вспомогательных средств разработчика и дополнительные информационные материалы.

Более подробно различия между этими двумя версиями представлены в табл. 3.8.

Таблица 3.8. Набор средств разработки программ в стандартной (С) и профессиональной (П) версиях Visual FoxPro

Средства разработки и наличие компонентов СУБД	С	П
Возможности для разработки программного обеспечения		
Визуальные средства разработки	+	++
Возможность создания собственных классов и подклассов, включая наследование, инкапсуляцию и полиморфизм	+	++
Наличие визуального отладчика прикладных программ	+	++
Возможность работы в операционных системах Windows 3.1n, Windows NT и Windows 95	+	++
Наличие большого набора элементов управления, позволяющих создать удобный пользовательский интерфейс	+	++
Возможность распространять прикладные программы, разработанные программистом для последующего использования в операционных системах Windows 3.1, Windows NT и Windows 95		+
Наличие Мастера установки (Setup Wizard), позволяющего создать набор дистрибутивных дисков для распространения пользовательского приложения		+
Наличие Class Browser - средства работы с классами		+
Возможность создания независимого запускаемого модуля пользовательского приложения в виде EXE-файла и его свободное распространение		+
Наличие дополнительных элементов управления OLE, которые могут быть использованы при разработке прикладной программы		+
Полное руководство программиста по Win32 API, позволяющее использовать стандартный набор и дополнительные DLL-функции (только на CD-ROM)		+
Доступ к данным и поддержка технологии клиент-сервер		
Встроенная поддержка реализации технологии "клиент-сервер" с наиболее популярными серверами баз данных, такими, как Microsoft SQL Server, ORACLE 6 и ORACLE 7, с помощью 32-битных драйверов ODBC	+	++

версии 2.0

Наличие визуальных средств проектирования базы данных, локальных и внешних просмотров	+ ++
Наличие Мастера создания таблиц в формате Microsoft SQL Server на основе данных в формате Visual FoxPro (Upsizing Wizard)	+
Поддержка технологии OLE 2.0	
Наличие элементов управления OLE Container, позволяющих во время работы программы использовать в ней компоненты OLE	+ ++
Поддержка OLE Automation, которая позволяет управлять другими приложениями из Visual FoxPro	+ ++
Поддержка редактирования на месте OLE-объектов, встроенных в Visual FoxPro без необходимости перехода в другое приложение	+ ++
Программы, документация и дополнительные средства разработчика	+ +
Возможность использования внешних API библиотек	+
Возможность разработки приложений, поддерживающих различные кодовые страницы для работы с данными,, хранящимися на различных языках	+
Наличие компилятора графических файлов контекстной помощи	+
Наличие программы GENDBC.PRG для генерации программы, которая может перестроить базу данных	+
Наличие более сотни файлов изображений и курсоров для использования в прикладной программе	+
Наличие средства создания и редактирования изображений, иконок и курсоров Imagedit	+
Наличие документации Developers Guide, Installation Guide, Quick Reference Guide и Users Guide	+ ++
Наличие документации Language Reference и Professional Features Guide	+

Почти сразу после выхода Visual FoxPro появилась и локализованная версия, в составе которой поставляется документация и файл контекстной справки на русском языке. Для максимально быстрого освоения новых приемов разработки пользовательского приложения программист может воспользоваться специальным пакетом - **Mastering Microsoft Visual FoxPro 3.0**, который содержит CD-ROM с интерактивной мультимедийной системой обучения на английском языке. Этот пакет содержит пояснительный текст, видеофрагменты и аудиокomentarий и предназначен для достаточно опытных программистов. Он охватывает следующие темы:

- Проектирование приложения, баз данных и таблиц.
- Объектная модель Visual FoxPro.
- Использование визуальных классов.
- Разработка форм, отчетов, запросов и представлений.
- Использование OLE и OLE Automation.
- Разработка приложений в архитектуре "клиент-сервер".
- Обработка ошибок и отладка пользовательского приложения.
- Особенности разработки приложения для работы в локальной сети.
- Создание меню и панелей инструментов.
- Работа с Project Manager, преобразование программ, разработанных на FoxPro 2.x в версию 3.0, словарь данных.
- Использование динамических библиотек Windows.
- Мастера и построители.

СУБД Access может быть приобретена как отдельный пакет, так и в составе пакета MS Office for Windows 95 Professional. Существует полностью локализованная версия Access, в которой, помимо документации и значительной части контекстной помощи, на русский язык переведен весь интерфейс. Русской версией Access, конечно, комплектуется и профессиональная русская версия

MS Office.

В составе MS Office for Windows 95 Professional кроме СУБД Access вы получите следующие хорошо интегрированные с ней пакеты:

- MS Excel - мощный пакет электронных таблиц для выполнения динамических расчетов и графического представления данных.
- MS Word - один из наиболее популярных для среды Windows текстовых процессоров с очень широкими возможностями.
- Power Point - программа для подготовки презентаций.
- Sheduler Plus - индивидуальный электронный планировщик и записная книжка.

Замечательным свойством всех пакетов, входящих в состав MS Office, является не только поддержка технологий OLE 2.0 и ODBC, но и предоставляемая программисту возможность использования общего языка программирования - Visual Basic for Application.

Для профессиональных разработчиков предназначен пакет Access Developer's Toolkit 95, позволяющий распространять разработанное приложение для поль-зователей и включающий следующие компоненты:

- Библиотека для работы с базами данных Access.
- Инструментарий для управления репликациями, включающий планировщик и поддержку исключительных соединений.
- Более 10 новых 32-разрядных элементов управления OLE.
- Расширенный Мастер для создания программы установки пользовательского приложения.
- Компилятор файла контекстной справки для Windows 95.
- Справка по Visual Basic.
- Справка по использованию объектов для доступа к данным.
- Справка по объектной модели MS Office.

Visual Basic поставляется в трех вариантах: стандартная и профессиональная версии, и версия масштаба предприятия (Enterprise Edition):

- Стандартная версия предназначена для программистов, не разрабатывающих коммерческих приложений, студентов и начинающих программистов.
- Профессиональная версия предназначена для опытных разработчиков коммерческих приложений. Она содержит, в частности, дополнительные возможности для работы с данными.
- Версия масштаба предприятия предназначена для групп разработчиков, занятых созданием приложений в архитектуре "клиент-сервер" для целого предприятия.

Более подробно различия между этими тремя версиями представлены в табл. 3.9.

Таблица 3.9. Набор средств разработки программ в различных версиях Visual Basic 4.0a - в стандартной (С), профессиональной (П) и в версии масштаба предприятия (МП)

Средства разработки	С	П	МП
Визуальная среда разработки, включающая мощный структурный язык программирования	+	++	++
Возможность расширения среды разработки за счет встроенного (Add-in) дополнительного инструментария разработчика (типа CASE-средств)	+	++	++
Возможность использования готовых компонентов для быстрой разработки приложений (ActiveX, 16-разрядные Visual Basic Custom Controls, OLE-объекты и DLL)	+	++	++
Редактор с цветным выделением синтаксиса и компилятор для получения псевдокода для 16-битной и 32-битной операционных систем на основе одного и того же кода	+	++	++
Отладчик программного кода с возможностью отслеживания значений переменных, отметки точек останова и сохранения стека вызова	+	++	++

подпрограмм			
Набор стандартных элементов управления	+	++	++
Элементы управления Windows 95, трехмерные элементы управления, элементы управления для мультимедиа и т. п.		+	++
Поддержка операционных систем	32- бит	16 и 32- бит	16 и 32- бит
Менеджер компонентов			+
Утилита построения профиля программы			+
Программа управления версиями Visual SourceSafe			+
Процессор данных (в стандартную версию Visual Basic включена ограниченная подверсия процессора данных, которая, например, позволяет получить доступ к существующей БД, но не позволяет создать новую)	+	++	++
Возможность использования данных Microsoft Access, FoxPro, dBASE, Paradox и Btrieve (только в 16-битной версии) наряду с таблицами Microsoft Excel и текстом	+	++	++
Менеджер данных для визуальной работы с данными	+	++	++
Элементы управления для работы с данными без необходимости написания кода	+	++	++
Полная поддержка стандарта ODBC 2.0,, включающая прокручивающиеся курсоры. Драйвер для Microsoft SQL Server	+	++	++
Data Explorer			+
Высокоэффективные элементы управления внешними данными (Remote Data Control - RDC), основанные на ODBC и оптимизированные для работы с ORACLE Server и Microsoft SQL Server			+
Поддержка OLE 2.0	+	++	++
Создание сервера OLE Automation	+	++	++
Утилита и Мастер создания программы для распространения и установки пользовательского приложения	+	++	++
Библиотека графических изображений, включающая более 450 значков для использования в пользовательском приложении	+	++	++
Утилита построения отчета Crystal Reports for Visual Basic 4.0 и соответствующие элементы управления для печати отчета в пользовательском приложении		+	++
Справка по Windows 3.1 и Windows 95 API (справка по Win32 API только на CD-ROM)		+	++
Более 250 изображений в форматах bmp и wmf		+	++
Компилятор файла контекстной помощи для пользовательского приложения		+	++

Так же как для Visual FoxPro, для Visual Basic существует записанная на CD-ROM интерактивная система изучения этого средства разработки - Mastering Microsoft Visual Basic 4.0. Этот пакет охватывает следующие основные темы:

- Обзор Visual Basic как средства быстрой разработки приложений.
- Контроль вводимых пользователем в прикладную программу значений.

- Обработка ошибок.
- Доступ к данным посредством элементов управления.
- Объекты для доступа к данным.
- Разработка баз данных.
- Подготовка отчетов с помощью **Cristal Reports**.
- Использование динамических библиотек **Windows**.
- Применение **OLE** и создание **OLE**-сервера.
- Создание вспомогательных средств разработки.
- Создание приложения, способного работать с электронной почтой.
- Позиционирование приложения для решения реальных задач в области бизнеса.

Microsoft SQL Server - мощное средство управления данными в архитектуре "клиент-сервер" поставляется как отдельный пакет и в составе пакета **BackOffice**. **MS SQL Server** поддерживает следующие сетевые протоколы:

- **Microsoft Windows NT Server**
- **Microsoft LAN Manager**
- **Novell NetWare**
- **TCP/IP**
- **IBM LAN Server**
- **Banyan VINES**
- **DEC PATHWORKS**
- **Apple AppleTalk**

Он также обеспечивает работу следующих клиентов:

- **Microsoft Windows 3.1**
- **Microsoft Windows 95**
- **Microsoft Windows for Workgroups**
- **Microsoft Windows NT Workstation**
- **Microsoft MS-DOS**
- **UNIX**
- **Apple Macintosh**
- **IBM OS/2**

В состав пакета **BackOffice** помимо **MS SQL Server** входят еще 4 хорошо интегрированных пакетов программ для эффективной работы в системах клиент-сервер:

- **Microsoft Windows NT Server** - это многоцелевая сетевая операционная система, включающая чрезвычайно быструю файловую систему, серверы приложений и печати, создающие основу для функционирования как самого **SQL Server**, так и пользовательских приложений. Эта операционная система способна функционировать на компьютере, имеющем до 32 процессоров, и легко интегрируется в самые распространенные сетевые системы, такие как **Novel NetWare**, **UNIX** и т. д.
- **Microsoft SNA Server** - это система (шлюз) для связи сети персональных компьютеров с мэйнфреймами (большими ЭВМ) фирмы **IBM**, использующими протоколы **SNA (Systems Network Architecture)**.
- **Microsoft Systems Management Server** позволяет выполнять администрирование компьютеров, распределенных в сети. Это предоставляет возможность централизованно проводить установку и модернизацию программного обеспечения на всех компьютерах, включенных в сеть, а также при необходимости выполнять их диагностику.
- **Microsoft Mail Server** - система передачи электронных сообщений, а также удобное средство организации и управления всем потоком входящих и исходящих сообщений. В состав пакета входит дизайнер электронных форм - **Microsoft Electronic Forms Designer**. Используя входящие в него шаблоны и язык программирования **Visual Basic**, пользователи смогут создавать собственные электронные формы и бланки и пересылать их друг другу.

Стоит добавить, что пакет **BackOffice** позволяет легко интегрировать все сетевые службы и ресурсы с помощью единой процедуры регистрации доступа пользователя. Все входящие в него программы отвечают требованиям секретности правительства США C2 и европейскому сертификату E3.

Перед выбором соответствующего средства разработки стоит хорошо подумать о том, на каких компьютерах и под управлением какой операционной системы будет работать создаваемое вами пользовательское приложение. Требования к программному и аппаратному обеспечению сведены в табл. 3.10.

Таблица 3.10. Требования к программному и аппаратному обеспечению

Показатели	Visual FoxPro 3.0	Access 7.0	Visual Basic 4.0	SQL Server 6.5
Минимальные требования к процессору компьютера	486DX	486DX	386DX	486DX-33 (Intel Pentium), PowerPC, MIPS, R4xxx или Alpha AXP
Операционная система (наиболее ранняя поддерживаемая версия):	+	+	+	+
Windows 3.1	+	++	++	
Windows 95	++	++	++	
Windows NT 3.5	+	++	++	++
Необходимый объем оперативной памяти, Мб	8 (12)	12 (16 для Windows NT)	6 (16 для Windows NT)	16 (32)
Занимаемый объем на жестком диске, Мб	15-80	10-40	8-36	80
Дополнительные устройства	Мышь, привод CD-ROM	Мышь, привод CD-ROM	Мышь, привод CD-ROM	Привод CD-ROM

Не меньшее значение, чем параметры, имеет качество техники, на которой планируется использовать программу. Даже средней сложности система автоматизации обработки данных заставляет компьютер работать с максимальной интенсивностью, используя все возможности процессора и пересылая громадные объемы данных между процессором, жестким диском и оперативной памятью. Особо следует обратить внимание на качество и надежность работы сетевых плат, если ваше приложение работает в локальной сети. Очень часто именно низкого качества сетевые платы приводят к разрушению баз данных и индексных файлов, что влечет за собой весьма длительные простои и потери данных.

Глава 4

Основы языка программирования

- 4.1. Что такое язык программирования
- 4.2. Как написать программу
- 4.3. "Горячая десятка"
- 4.4. Еще несколько навязчивых советов

Авторы, обладая гипертрофированным самомнением, полагают, что эту книгу будут читать люди с самой разной степенью компьютерной подготовки. Возможно даже, что она попадется тем, кто слышал о программировании, как о занятии достаточно странных, обычно бородатых людей, бормочащих о каких-то нулях и единицах. На самом деле этот страшный образ порожден довольно давно, и истории про нули и единицы не имеют непосредственного отношения к

современным СУБД для персональных компьютеров, которые оснащены языковыми средствами, позволяющими достаточно легко написать простую программу.

В этой главе мы попробуем дать основные сведения о языке программирования Visual FoxPro читателям, которые от работы с диалоговыми средствами хотят перейти к написанию пользовательской программы. Для тех, кто уже пишет такие программы, мы систематизировали справочный материал и дали несколько советов по наиболее эффективным приемам программирования в рассматриваемых средствах разработки.

4.1. Что такое язык программирования

В [предыдущей главе](#) мы познакомились с основными действиями, которые можно выполнить с помощью диалоговых средств в визуальной среде разработки. Очевидно, что, работая с СУБД, некоторые действия приходится выполнять многократно. Например, открывать одни и те же таблицы. Многие функции работы пользовательского приложения невозможно реализовать, используя только визуальные средства. Решить эти проблемы можно с помощью языка программирования.

В этом параграфе вы узнаете:

- Из чего состоит язык программирования.
- Где могут храниться нужные данные.
- С помощью каких средств можно выполнять какие-либо действия с данными.
- Как делятся в программе переменные и массивы по области действия.

При попытке описания языков программирования в рассматриваемых средствах разработки авторы столкнулись с достаточно большими трудностями. Не сомневаясь в своих интеллектуальных способностях, вину за это они целиком и полностью возложили на сами языки, а пособиями признали тех, кто их придумал - разработчиков из Microsoft.

Действительно, так как СУБД Access использует язык программирования Visual Basic, то остается рассказать о нем и языке программирования Visual FoxPro. Эти языки программирования имеют достаточно много общих черт, но одна из них - богатое историческое наследие - привела к тому, что современные объектно-ориентированные свойства в них сосуществуют с традиционными структурными составляющими. Причем число команд и функций, составляющих структурную основу языка, перевалило далеко за тысячу. Чтобы разобраться в этой лавине, начнем со структурной части рассматриваемых языков, а в [следующей главе](#) изучим их объектно-ориентированные возможности. При этом мы будем стараться максимально выделять общие черты рассматриваемых языков программирования и заранее приносим свои извинения опытным разработчикам за игнорирование каких-то, может быть, и достаточно важных особенностей каждого языка. Например, в наследство от Xbase в Visual FoxPro до сих пор можно символьные значения указывать не только в кавычках, но и в квадратных скобках. В Visual Basic так делать нельзя. Мы думаем, что стоит придерживаться общих возможностей и указывать символьные значения в кавычках, не упоминая о квадратных скобках.

Язык программирования представляет собой набор команд, которые последовательно обрабатываются интерпретатором и преобразуются им в машинный код, в свою очередь обрабатываемый микропроцессором. С помощью команд мы выполняем какие-либо действия, аналогично выбору команды в меню. Типичная структура команды:

COPY TO *FileName* [**FIELDS** *FieldList*] [*Scope*][**FOR** *lExpression*]

Название команды является ключевым элементом для ее идентификации. В связи с тем, что разработчики языка старались дать названиям команд максимальную смысловую нагрузку для их более легкого запоминания, в ряде случаев команды получились достаточно громоздкими. Что ж, если вы не хотите долго стучать по клавиатуре, откроем маленький секрет. Например, в Visual FoxPro названия команд в большинстве случаев можно сокращать до четырех символов. Если в каком-то случае так делать нельзя из-за опасности потерять уникальность идентификации, об этом обязательно будет написано в справочном файле Visual FoxPro при описании данной команды.

Опции служат для уточнения действия, выполняемого командой. В данном примере можно использовать опцию **FIELDS** для указания списка тех полей, которые будут копироваться в новый файл.

Часть команды, обозначенная словом *Scope*, позволяет задать диапазон записей, на которые будет воздействовать команда. Если мы используем эту возможность, то в команде вместо слова *Scope* надо использовать один из перечисленных вариантов: **ALL** - все записи в таблице; **NEXT *nRecords*** - указанное число записей после текущей (включая текущую); **RECORD *nRecordNumber*** - запись с указанным номером; **REST** - записи от текущей до конца таблицы.

Условие выполнения команды FOR позволяет предопределить выполнение команды в зависимости, например, от содержания данных в полях.

Для обработки информации мы можем использовать данные, хранящиеся в таблицах или в памяти компьютера. В последнем случае для данных в памяти, которые называются переменными, мы должны определить идентификатор, ссылаясь на который можно однозначно установить, какие данные мы имеем в виду. Для хранения в памяти однородных данных используются массивы.

Таблица 4.1. Способы представления данных

Способ представления	Описание
Константы	Единичные элементы данных, записываемые в программном коде и неизменяемые в процессе работы
Переменные	Единичные элементы данных, хранящиеся в оперативной памяти (ОЗУ)
Массивы	Множество элементов данных, хранящихся в ОЗУ
Записи в таблицах	Множество строк, содержащих заранее определенные поля, каждое с предопределенным фрагментом данных, хранящихся в файле таблицы

Константы часто используются для включения в выражения каких-то предопределенных данных, которые не изменяются во время работы программы. В качестве констант мы можем использовать данные различных типов:

- Символьные данные записываются в кавычках. Например, мы можем запомнить слово *имя*, используя его указание в кавычках: "Имя".
- Данные типа "дата" или "дата и время" в Visual FoxPro записываются в фигурных скобках: {10/10/95}. В Visual Basic функции этих двух типов данных выполняет один тип данных "дата и время". Данные этого типа выделяются значками "решетка": #10/10/95#.
- Числовые данные используются без каких-либо разделителей.
- Логические данные в качестве константы могут принимать одно из двух значений. В Visual FoxPro при записи они ограничиваются точками: ".Т." - соответствует истине (True), ".F." - ложному значению (False). В Visual Basic присваивание любого отличного от нуля значения установит константу в значение True, и только 0 - в значение False.

Для работы с данными используются операторы, перечень которых приведен в табл. 4.2. Для того чтобы вспомнить обозначение типа данных, вернитесь к табл. 3.2.

Помните, что с одним оператором необходимо использовать данные одного типа!

Многие из приведенных в табл. 4.2 операторов знакомы нашим читателям со школьной скамьи. Для некоторых же из этих операторов требуется более пространное пояснение.

- Оператор "\$" используется для поиска символа или набора символов в каком-либо символьном выражении или поле примечаний. Искомый символ указывается слева от этого оператора.
- Оператор "==" отнюдь не то же самое, что оператор "=". Если последний можно идентифицировать словом "равно", то для первого больше подходит название "идентично". Например, если мы будем искать данные по условию cName = "ИВАН", то Visual FoxPro будет считать верным результат, если найдет значения "ИВАНОВ", "ИВАНЕНКО", так же как и указанное слово "ИВАН". Если же в выражении для поиска указать оператор "==", то к верному результату приведет только найденное слово "ИВАН". Это различие очень удобно использовать для поиска данных по неполному соответствию, когда точно неизвестно искомое значение.

Поля, переменные, константы, функции и операторы представляют собой элементы для составления выражений. При написании выражений необходимо учитывать приоритет выполнения операций:

1. Возведение в степень

2. Умножение и деление
3. Сложение и вычитание
4. Сложение символьных выражений
5. Операции сравнения
6. NOT
7. AND
8. OR

При необходимости изменения порядка действий следует использовать скобки.

Переменные в программе имеют строго определенные области действия и по этому признаку делятся на три типа.

Локальные переменные (*private*) существуют только во время работы процедуры или программного файла (модуля), в котором они были определены. По умолчанию всем переменным, определяемым в программе, присваивается статус локальных. Явно определить статус переменной в Visual FoxPro можно командой

PRIVATE *MemVarList* | **PRIVATE ALL** [**LIKE** *Skeleton* | **EXCEPT** *Skeleton*]

В Visual Basic для этого используется команда

Dim *VarName* [(*Subscripts*)] [**As** [**New**] *Type*][, *VarName* [(*Subscripts*)] [**As** [**New**] *Type*]]...

или

Static *VarName* [(*Subscripts*)] [**As** [**New**] *Type*][, *VarName* [(*Subscripts*)] [**As** [**New**] *Type*]]...

В последнем варианте объявленные переменные сохраняют свои значения до конца работы программы.

Важная особенность локальных переменных заключается в том, что если они были определены с теми же именами, которые были объявлены ранее (в программе более высокого уровня), то после завершения работы программы "старые" значения будут восстановлены. Таким образом, локальные переменные, созданные в какой-либо процедуре, будут действовать и во всех вызываемых из нее процедурах и функциях, но перестанут быть доступными, как только мы возвратимся из создавшей их процедуры на более высокий уровень. Опции команды в Visual FoxPro позволяют не приводить полный список всех переменных, а использовать для их идентификации шаблон, который с помощью символов "*" и "?" укажет распространение действия команды на переменные, имена которых соответствуют шаблону (**LIKE**), или на переменные, имена которых не соответствуют приведенному шаблону (**EXCEPT**). Напомним, что шаблон обычно включает общую часть имени и знаки замещения "?" и "*", первый замещает один алфавитно-цифровой символ, а второй - любое их число. Это позволяет знаком "*" задавать все имена, а несколькими знаками "?" имена соответствующей длины.

В Visual Basic мы должны перечислить все объявляемые переменные.

При большом количестве однотипных переменных более эффективно использовать вместо них массивы, то есть одно- или двумерные таблицы переменных под общим именем. Обращение к элементам массива производится с помощью указания после имени массива в круглых или квадратных скобках нужных номеров строк и (или) столбцов. Перед использованием массивов в Visual FoxPro их надо объявить командой

DIMENSION *ArrayName1*(*nRows1* [, *nColumns1*]) [, *ArrayName2*(*nRows2* [, *nColumns2*])] ...

Каждый массив может содержать не более 65000 элементов, то есть предельная размерность двумерного массива может составить 254x255, 1000x65 и т. п. В последнем случае *nRows1* = 1000 и *nColumns1* = 65, если значение *nColumns1* не определено, то массив одномерный. После объявления массива по умолчанию всем его элементам присваивается логический тип данных со значением .F..

В Visual Basic задать массив можно в команде **Dim**, указав его размерность в параметре *Subscripts*.

Для задания нужного типа данных, а это относится и к переменным, в Visual FoxPro необходимо использовать команду

STORE *Expression* **TO** *MemVarList* | *ArrayList*

или использовать присвоение

MemVar / Array = Expression

= При указании идентификатора массива без аргументов указанное значение *Expression* присваивается всем элементам массива. Например:

```
DIMENSION aMembers(4,2)
STORE 5 TO aMembers
```

В Visual Basic такая же процедура будет выглядеть следующим образом (нам придется использовать цикл, о чем мы будем говорить в дальнейшем):

```
Dim aMembers(4,2), nX As Integer
For nX = 0 To 4
  aMembers(nX) = 5
Next nX
```

Для определения переменной типа даты в Visual FoxPro можно использовать следующую команду, которая одновременно присвоит ей нулевое значение:

```
STORE CTOD(' . ') TO dDate
```

или

```
STORE {} TO dDate
```

Для присвоения конкретного значения лучше воспользоваться вариантом с фигурными скобками, например:

```
SET DATE GERMAN
STORE {01.01.91} TO dDate
```

Для форматирования значения даты в Visual FoxPro удобно использовать установочную команду **SET DATE**.

В Visual Basic присвоение значения даты можно выполнить следующим образом:

```
Dim dToday As Date
dToday = #1/11/96#
```

А для форматирования даты надо использовать функцию

```
Format(Expression [, Format [, FirstDayOfWeek [, FirstWeekOfYear]])]
```

Например:

```
dDate = Format(dToday, "dd.mm.yy")
```

Возвращает значение переменной dToday в виде 01.11.96.

Глобальные переменные (public) будут сохранять свои значения во всех программных файлах и вызываемых ими процедурах. Для объявления переменных и элементов массива глобальными в Visual FoxPro используется команда

```
PUBLIC MemVarList | [ARRAY] ArrayName1(nRows1 [, nColumns1]) [, ArrayName2(nRows2 [, nColumns2])]...
```

В Visual Basic такая команда будет выглядеть аналогично команде **Dim**

```
Public VarName [(Subscripts)] [As [New] Type][, VarName [(Subscripts)] [As [New] Type]] ...
```

Лишь после объявления глобальным переменным можно присвоить какие-либо значения. *Внутренние переменные (local)* действуют только в пределах процедуры или функции, в которых были созданы. К ним нельзя обратиться из программы или функции ни более высокого, ни более низкого уровня. Объявить переменные внутренними в Visual FoxPro можно командой

LOCAL MemVarList | **[ARRAY] ArrayName1(nRows1 [, nColumns1])** [, **ArrayName2(nRows2 [, nColumns2])**] ...

В Visual Basic для этой цели можно использовать команду **Dim**.

Вы должны сначала объявить переменные или массивы внутренними и только потом присвоить им необходимые значения. После выхода из процедуры или функции, в которой были созданы внутренние переменные, они будут удалены из памяти.

Региональные переменные или массивы (regional) могут использоваться только в Visual FoxPro. Они подобны локальным и объявляются с помощью команды

REGIONAL MemVarList

Перед командой помещается директива

#REGION nRegionNumber

с указанием номера региона *nRegionNumber* (от 0 до 31), в котором действуют переменные, перечисленные в списке *MemVarList*. В регионе с другим номером та же переменная может иметь другое значение.

4.2. Как написать программу

В этом параграфе мы научимся составлять очень простые программы, которые будут работать в

- СУБД *Visual FoxPro*;
- *Visual Basic*;
- СУБД *Access*.

Писать даже самую простую программу непросто. Из рассматриваемых средств разработки легче всего сделать это в Visual FoxPro. С него и начнем.

Чаще всего самым трудным бывает первый шаг. Не будем искать трудностей и просто избежим этого первого шага. Поручим это самой СУБД, пусть сама напрягает свои "электронные мозги".

Обратите внимание, что при выполнении каких-либо действий в интерфейсе Visual FoxPro в окне *Command* автоматически записывается соответствующая команда. Если несколько последовательных команд перенести в программный файл, то получится программа. Если ее запустить, несколько действий, которые мы до этого делали с помощью меню и соответствующих диалоговых окон, будут выполнены сразу без нашего непосредственного вмешательства.

Если мы захотим из таблицы SPPOS.DBF, содержащей список поставщиков, прочитать данные, относящиеся к фирме с названием "Компьютерное образование", мы должны из меню *File* выбрать команду *Open*, установить тип файла и из списка выбрать SPPOS.DBF. Из меню *Database* выбрать команду *Browse* и найти запись, относящуюся к указанной фирме.

Программно тот же результат можно получить, набрав в окне *Command* следующие команды:

```
USE Sppos
BROWSE
LOCATE FOR Company = "Компьютерное образование"
```

Если приведенная выше последовательность команд будет записана в файле, мы получим простейшую программу. Это может быть сделано несколькими способами. Например, можно из меню *File* выбрать команду *New*, установить тип файла *Program*. Откроется окно Редактора FoxPro. В нем можно набрать указанные строчки или скопировать их из окна *Command*. При закрытии окна редактирования будет необходимо указать имя файла, которое в дальнейшем и будет являться именем программы.

Для изменения или дополнения программы необходимо ее открыть с помощью команды *Open* в меню *File*.

Если вы взялись за разработку программ, то вам часто придется проделывать эту операцию и работать с редактором Visual FoxPro. Поэтому уделим ему немного внимания. Начнем с конфигурации, ведь от того, насколько удобно установлены те или иные параметры работы редактора, зависит производительность вашего труда.

Откройте какой-либо файл с программой или создайте новый файл. Выберите команду *Options* в меню *Tools*. В появившемся диалоговом окне найдите вкладку *Edit*. Доступные установки для редактора видны из рис. 4.1.



Рис. 4.1.

При первом запуске программы и в последующем, при ее изменении, FoxPro компилирует исходный код программы в так называемый препроцессорный код, или "р-код", который выполняется во много раз быстрее интерпретируемой программы. Файлы с "р-кодом" получают другое расширение. Обратите внимание на очень удобную возможность компиляции программы перед ее сохранением. Включение этой опции позволяет сразу определить синтаксические ошибки, выявляемые на этом этапе, - недостаток или отсутствие запятых, неправильное написание ключевых слов и т. д.

В связи с тем, что редактор Visual FoxPro используется не только для работы с файлами программ, но и вообще для редактирования текстовых данных, например в полях примечаний, то перечень доступных опций в диалоговом окне *Options* зависит от типа окна, которое было активным перед его вызовом. Например, если активным окном было окно с данными поля примечаний, то внешний вид окна *Options* будет таким, как это видно на рис. 4.2. Естественно, при этом некоторые опции, например компиляции, будут недоступны.

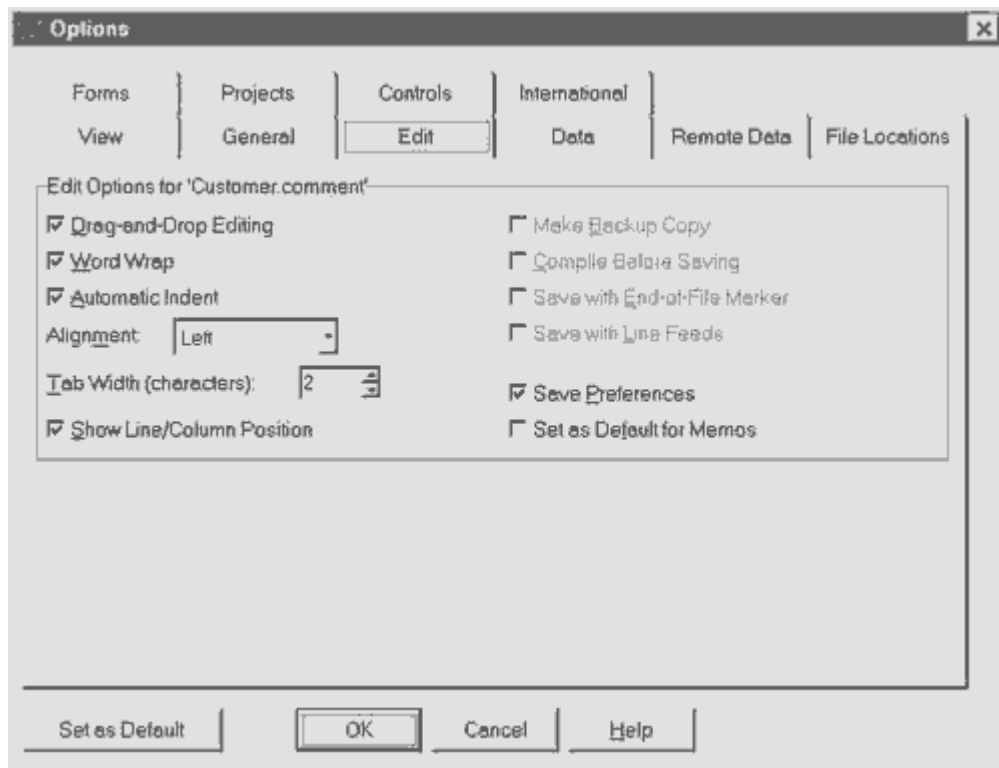


Рис. 4.2.

При работе с редактором мы можем использовать команды, которые располагаются в меню *Edit* и *Format*. Еще удобнее использовать клавиатурные комбинации, перечень которых приведен в табл. 4.3. Не забудьте также о кнопках на стандартной панели инструментов. При наборе программы в редакторе очень удобно использовать макросы.

Макросом называется предварительно записанная последовательность команд, которая выполняется при его запуске. Для запуска макроса чаще всего используются клавиатурные комбинации.

В Visual FoxPro макрос можно записать с помощью соответствующей команды в меню *Tools*. Появляющееся при этом диалоговое окно представлено на рис. 4.3.

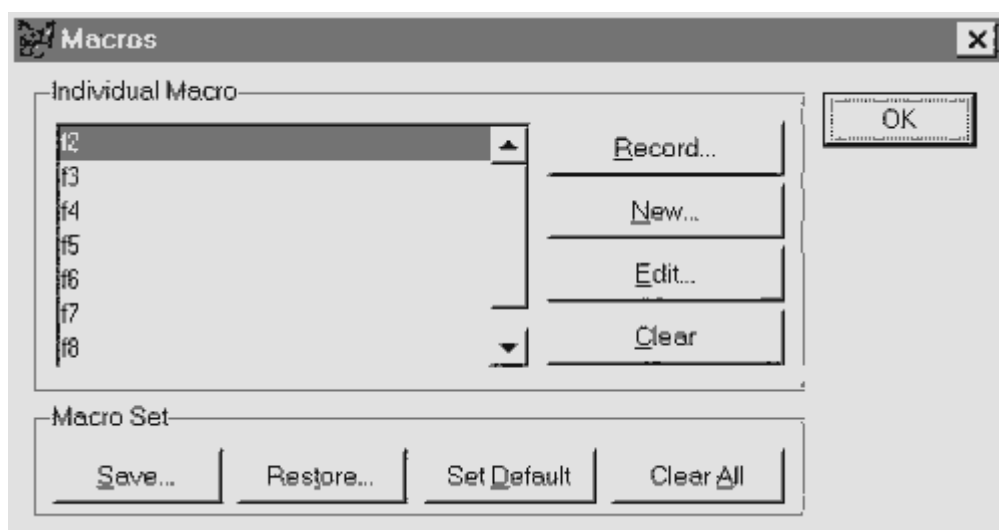


Рис. 4.3.

В своей работе вы можете использовать несколько наборов макросов, сохраняя их в соответствующих файлах. Для сохранения и загрузки необходимых макросов надо использовать

кнопки, расположенные внизу диалогового окна.

Для записи макроса нажмите кнопку New. Появится диалоговое окно для создания и редактирования макросов, представленное на рис. 4.4. Нажмите клавиатурную комбинацию, с помощью которой вы хотите в дальнейшем вызывать данный макрос.

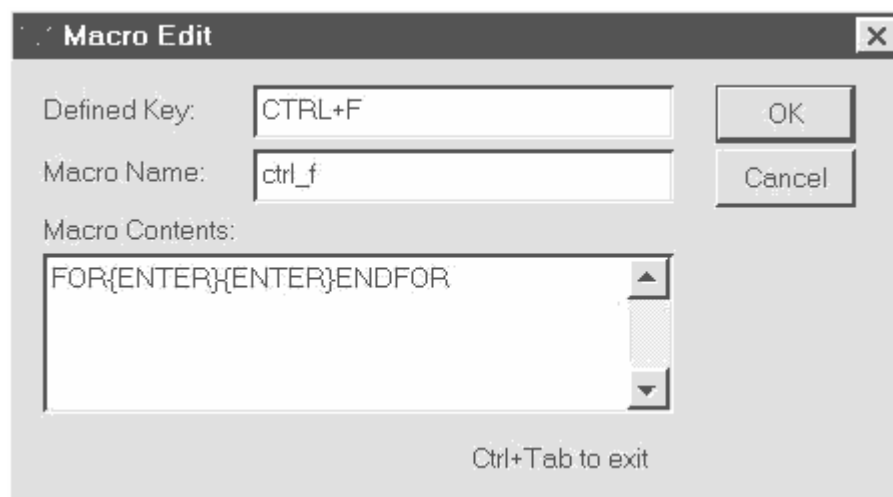


Рис. 4.4.

Таблица 4.3. Комбинации клавиш, используемые при работе с редактором

Комбинация клавиш	Пункт меню	Выполняемое действие
<i>Ctrl+A</i>	Select All	Выделить весь текст в окне редактора
<i>Ctrl+C</i>	Copy	Скопировать выделенный текст
<i>Ctrl+F</i>	Find	Найти
<i>Ctrl+L</i>	Replace	Заменить
<i>Ctrl+R</i>	Redo	Вернуть отмененное действие
<i>Ctrl+V</i>	Paste	Вставить фрагмент текста из буфера в место нахождения курсора
<i>Ctrl+W</i>	-	Заккрыть редактор и сохранить все сделанные изменения
<i>Ctrl+X</i>	Cut	Вырезать выделенный текст
<i>Ctrl+Z</i>	Undo	Отменить сделанное действие
<i>Ctrl+Home</i>	-	Переместить курсор в начало текста
<i>Home</i>	-	Переместить курсор в начало строки
<i>Ctrl+End</i>	-	Переместить курсор в конец текста
<i>End</i>	-	Переместить курсор в конец строки

В качестве примера мы с вами создадим макрос для автоматического набора какой-нибудь сложной команды. Пусть это будет команда задания цикла **FOR...NEXT**. Такие структурные команды чаще всего вызывают ошибки, так как тело цикла может занимать десятки строк и, разбираясь в его перипетиях, программист забывает о необходимости закрытия цикла. Присвоим этому макросу клавиатурную комбинацию Ctrl+F. Тогда по умолчанию он получит имя ctrl_f. Не следует пытаться присвоить макросу клавиатурную комбинацию, которая уже используется в Visual FoxPro. Теперь в поле Macro Contents наберем такой текст:

FOR{ENTER}{ENTER}NEXT

В фигурных скобках мы указали название клавиш, нажатие которых нам надо имитировать в данный момент. В этом примере мы имитируем двухкратное нажатие клавиши *Enter* для перехода на следующую строку и тем самым создаем внутри команды пустую строку для записи тела

цикла. Как правильно записать в макросе названия других клавиш и их сочетаний можно узнать, изучив длинную таблицу в справке к команде **ON KEY LABEL** после вызова контекстной помощи.

Теперь, работая в редакторе, мы всегда можем нажать клавиши Ctrl+F и в программе у нас автоматически появятся следующие три строки:

FOR
NEXT

Программа в FoxPro - это текстовый файл, содержащий набор команд, написанных в соответствии с синтаксическими правилами языка. Программа может иметь подпрограммы (процедуры), в которые помещаются часто повторяющиеся фрагменты кода, размещаемые после основного текста программы или в отдельном файле. Подпрограмма начинается с ключевого слова

PROCEDURE ProcedureName

и выполняется, пока не будет выполнено одно из следующих условий:

- Еще раз встретится слово **PROCEDURE**.
- Будет обнаружена команда **RETURN** - возвращение в предыдущую программу.
- Будет выдана команда **CANCEL** - прерывание работы программы.
- Будет выдана команда **QUIT** - выход из СУБД.
- Встретится новая команда **DO** для запуска другой программы.
- Будет достигнут конец файла.

В FoxPro аналогично подпрограмме трактуется понятие пользовательской функции, которая начинается с ключевого слова

FUNCTION FunctionName

и в отличие от процедуры, может вернуть необходимые значения в вызывающую программу. Имеются четыре способа вызвать функцию:

1. Присвоить возвращаемое значение переменной. Например, следующая строка кода запоминает текущую системную дату в переменной dToday:
dToday = DATE()
2. Включить вызов функции в команду. Например, следующая команда устанавливает по умолчанию каталог, имя которого возвращает функция **GETDIR()**:
SET DEFAULT TO GETDIR()
3. Напечатать возвращаемое значение в активное окно:
? TIME()
4. Вызвать функцию без запоминания где-либо возвращаемого значения:
SYS(2002)

Для прерывания выполнения программы служит оператор
RETURN [Expression | TO MASTER | TO ProgramName]

который возвращает управление в вызывающую программу, и в ней выполняется следующая команда после вызывающей; если указана опция **TO MASTER**, то управление возвращается на самый верхний уровень вызывающей программы, а эта же программа с опцией **TO ProgramName** передает управление в указанную программу. При использовании в функции команда автоматически возвращает .T., если не указано другое выражение на месте Expression.

RETRY

действует подобно команде **RETURN**, но при возвращении управления в вызывающую программу повторяется выполнение последней команды.

Создайте новый программный файл и наберите в редакторе приведенный ниже текст. В этой программе объявляется массив, элементы которого принимают значения от 1 до 10. Каждое присвоение значения сопровождается появлением на экране сообщения с указанием присвоенной величины. Посмотрите на сообщения, выводимые этой программой на экран.

* Пример простейшей программы
DIMENSION aSampleArray(10)

```

FOR nItem = 1 TO 10
    aSampleArray(nItem) = nItem
    = Append_proc()
NEXT
FUNCTION Append_proc
? "В массив добавлено новое значение "
?? nItem

```

Если эта программа показалась вам слишком примитивной, то без сомнения вы правы. По мере чтения книги и расширения лексического запаса языка (набора команд и функций), вы сможете сочинять куда более полезные вещи.

Теперь посмотрим на **Visual Basic**. Это средство разработки настолько сфокусировано на визуальных возможностях разработки программ, что написать вручную несколько строк кода, запустить их на выполнение и тут же увидеть результат не так просто для начинающего программиста, как в **Visual FoxPro**. Однако попробуем это сделать.

Начнем с задания команды *Options* меню *Tools*. В появившемся диалоговом окне настройки конфигурации **Visual Basic** выберем вкладку **Project** (рис. 4.5). В раскрывающемся списке **Startup Form** вместо значения, установленного по умолчанию, - **Form1**, выберем **Sub Main**. Значение **Form1** обеспечивает запуск в качестве главной программы визуальной формы с этим именем по умолчанию. Значение **Sub Main** позволяет в качестве запускаемой программы использовать процедуру.



Рис. 4.5.

Создадим в проекте программный модуль путем выполнения команды *Module* в меню *Insert*. Потом в этом же меню выполним команду *Procedure*. В появившемся диалоговом окне,

показанном на рис. 4.6, напомним имя создаваемой процедуры - **Main**, для типа процедуры выберем **Sub**, а для диапазона действия - **Public**. После нажатия кнопки **OK** в окне программного модуля появится шаблон, готовый для написания программного кода.

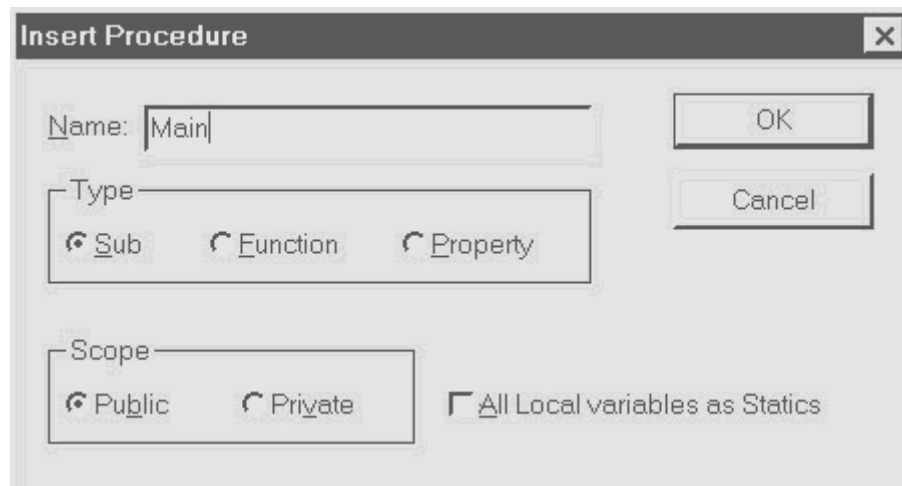


Рис. 4.6.

Здесь следует заметить, что редактор в **Visual Basic** по своим функциональным возможностям и принципам работы с текстом программы похож на редактор **Visual FoxPro**. В **Visual Basic** нет макросов, зато редактор обеспечивает выделение цветом ключевых слов, строк с ошибками и т. д. Цвет в редакторе существенно помогает избежать случайных ошибок. При этом вы можете сами устанавливать цвет для различных ситуаций, как это видно из рис. 4.7, на котором показано диалоговое окно **Options**, открытое на вкладке **Editor**.

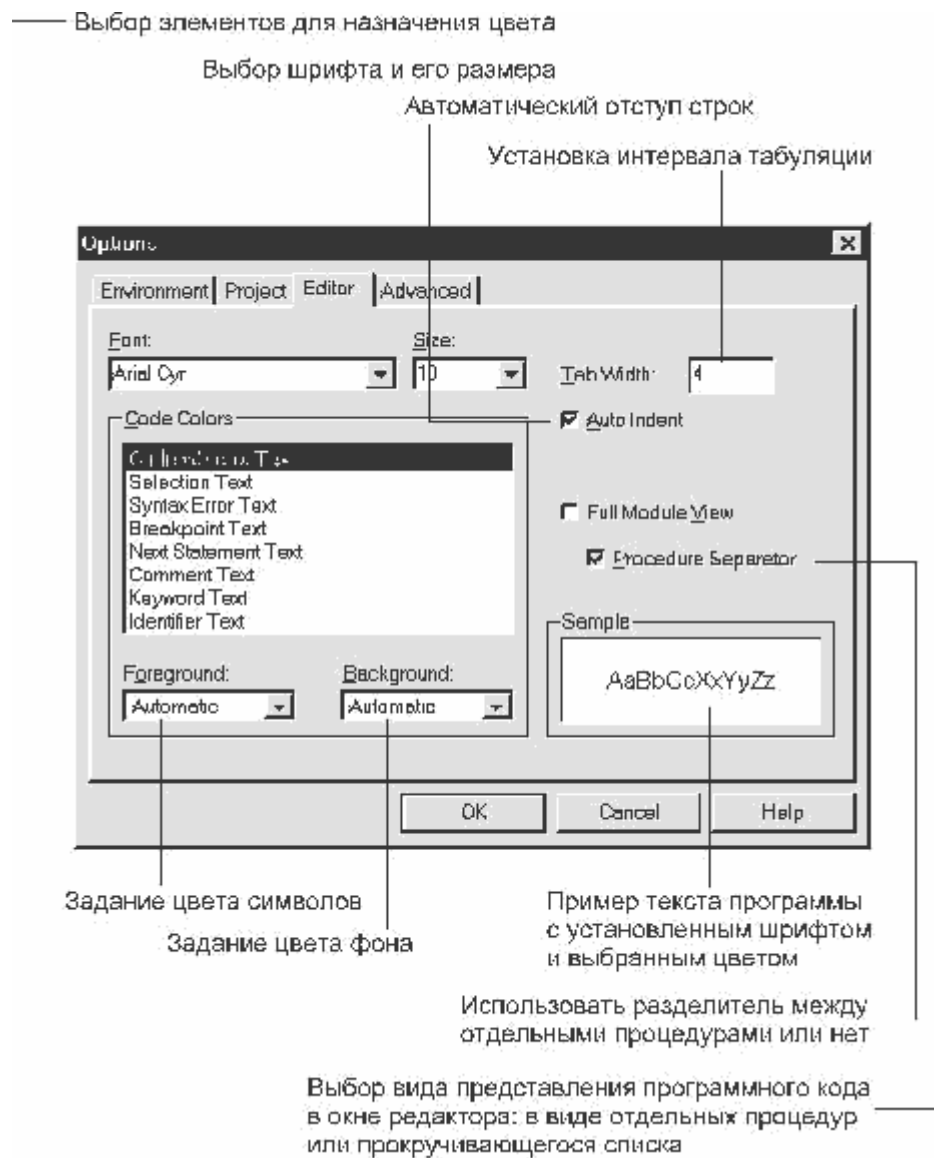


Рис. 4.7.

Теперь мы можем попробовать создать простейшую программу по аналогии с той, которую мы написали в Visual FoxPro. В окне программного модуля в раскрывающемся списке Proc выберите пункт (declarations), как это показано на рис. 4.8.



Рис. 4.8. Пример простейшей программы в Visual Basic

Наберите следующую строчку для объявления используемого массива и переменной как целых чисел:

```
Dim aSampleArray(10), nItem As Integer
```

Переключитесь в раскрывающемся списке Proc на пункт Main и наберите следующий текст:

```
Public Sub Main()
For nItem = 1 To 10
aSampleArray(nItem) = nItem
Append_proc
Next
End Sub
```

Теперь создадим новую процедуру с помощью команды *Procedure* меню *Insert* и в ней запишем код для функции, которая будет отображать процесс заполнения массива:

```
Private Function Append_proc()
Debug.Print nItem
End Function
```

Запустим нашу программу на выполнение командой *Start* в меню *Run*. Можно воспользоваться и соответствующей кнопкой на панели инструментов или просто нажать клавишу *F5*.

В отличие от Visual FoxPro в Visual Basic нельзя направить вывод результатов выполнения программы в главное окно. Его просто нет. Сначала надо создать какое-то окно, обычно в виде формы, а затем уже в него направлять вывод данных. Чтобы не усложнять нашу задачу, мы сознательно выбрали простейший вариант, используя для вывода окно отладки (*Debug Window*). Для этого перед командой *Print* необходимо указать через точку *Debug*. Для того чтобы увидеть результат, откройте окно отладки командой *Debug Window* в меню *View*.

Обратите внимание, что коды программ в Visual FoxPro и в Visual Basic очень похожи; код Visual Basic может легко читаться программистом, знающим только Visual FoxPro, и наоборот. В то же время существуют и определенные различия, которые с увеличением сложности программы разрастаются. Посмотрите на команды вызова функции и объявления переменных - в Visual Basic в отличие от Visual FoxPro мы должны явно объявить все используемые переменные и указать их тип.

Что касается СУБД Access, то, как мы уже говорили, многие воспринимают ее как типичное средство создания личных информационных систем, предусматривающее использование только имеющихся в Access диалоговых средств. Поэтому пользователи часто останавливаются на результатах, достигнутых с помощью визуальных средств разработки, и считают, что самостоятельное создание таблиц, не говоря уже о создании форм и отчетов, это нечто из ряда вон выходящее. На самом деле все не совсем так, и стоит относиться к Мастерам как к

чернорабочим, которые строят каркас здания, а отделкой лучше заниматься самостоятельно.

Программистов при переходе к Access из процедурного языка или среды с командной строкой обычно удручает отсутствие непосредственной возможности написать команду или процедуру, создать базу данных и при желании добавить тем же путем таблицы, формы или отчеты. При первом запуске Access этой возможности действительно нет. Вы не можете написать ни строки кода, пока не создадите базу данных или не откроете уже созданную.

Вы можете создать какую-либо процедуру после открытия контейнера БД. Для создания программ в Access используется язык программирования Visual Basic, о котором мы уже говорили. Перейдите на страницу Модули контейнера базы данных и создайте новый модуль. Меню Access слегка видоизменится. Надо привыкнуть, что меню динамически меняется, то есть при смене объекта, с которым вы работаете, загружается новое меню.

Теперь можно набрать следующую строку

Sub Ясоздаюбазуданныхвручную()

Не забудьте, что каждая процедура должна заканчиваться командой *End Sub*, но набирать эту строку не обязательно - после нажатия клавиши *Enter* она появится автоматически.

Ваша процедура должна выглядеть следующим образом:

```
Sub ЯсоздаюБазуДанныхВручную()
Dim MyDb As DATABASE
Set MyDb = _DBEngine.Workspaces(0).CreateDatabase("МояБазаДанных", _dbLangCyrillic)
End Sub
```

Чтобы запустить эту процедуру на исполнение, выведите на экран окно отладки и наберите в нем название процедуры, возможно, у вас она будет называться по-другому. После этого можете смело нажимать клавишу ввода, уверяем, что база данных у вас появится.

Для того чтобы в программе можно было разобраться хотя бы на следующий день после написания, не ленитесь писать комментарий к выполняемым действиям. Для написания строк комментария в Visual FoxPro надо в начале строки поставить "звездочку"

*** [Comments]**

Знак звездочки часто используется также для разделения отдельных смысловых фрагментов программы. Строки комментария нельзя вставлять внутрь команды, если, например, она написана на нескольких строках с использованием символа переноса - точки с запятой.

При написании комментария на одной строке с командой текст команды и комментария разделяется двумя амперсандами:

&& [Comments]

Эти знаки можно опустить только после команд, начинающихся на **END**.

В Visual Basic признаком комментария является знак апострофа:

[Comments]

Он может применяться и в начале строки, и после команды на одной строке с ней.

Признаком переноса командной строки в Visual Basic является знак подчеркивания, а в Visual FoxPro - точка с запятой.

4.3. "Горячая десятка"

Если вы следовали руководящим указаниям авторов, то вам не трудно написать простейшую программу. Если вы написали даже очень простую программу, то вы, несомненно, можете сказать, что являетесь программистом. Если вы являетесь программистом, то пришла пора заглянуть в красивую коробку, в которой, мы надеемся, вы принесли от продавца Access, Visual Basic или Visual FoxPro, и вместо Руководства пользователя вытащить оттуда Руководство программиста. Если после всего этого у вас упало настроение и стало рябить в глазах от длинных списков многочисленных объектов, команд и функций, не надо впадать в отчаяние.

В этом параграфе мы попробуем взять реванш у разработчиков Microsoft за их неумную фантазию. Возьмем на вооружение успехи лингвистов в деле составления частотных словарей, вообразим себя обладателями опыта составителей музыкальных хит-парадов и составим "горячую десятку" - расскажем о тех командах и функциях, которые сразу продвинут вас в деле написания программ далеко вперед.

10. Пока мы учимся программировать, нам очень поможет самая простая команда вывода данных в Visual FoxPro

? | ?? Expression

Мы даем здесь не полный ее синтаксис, так как эта команда вряд ли пригодится вам для чего-то более виртуозного, чем вывод нужного значения в процессе разработки и отладки прикладной программы. Ее очень удобно набирать в окне *Command*. Один знак вопроса всегда выводит значение выражения *Expression* с новой строки, два знака вопроса - на той же строке. В Visual Basic этой команде в наибольшей степени соответствует конструкция

Debug.Print Expression

которая выводит значение *Expression* в окно отладки **Debug Window**.

9. Для запуска программы или для передачи управления другой программе в **Visual FoxPro** дайте команду **Do** из меню **Program** и выберите файл с нужным именем, или в окне **Command** наберите команду

DO ProgramName1 | ProcedureName [WITH ParameterList] [IN ProgramName2]

Если в *ProgramName1* расширение не указывается, то **Visual FoxPro** будет пытаться запустить эту версию программы в следующей последовательности расширений для файла с одним и тем же именем:

- EXE - исполняемая версия.
- APP - прикладная программа.
- FXP - откомпилированная версия.
- PRG - программа.

Опция **WITH** используется для передачи параметров (заранее определенных данных) в программу (число параметров не должно превышать 27). По умолчанию параметры передаются по ссылке, для передачи по значению необходимо установить **SET UDFPARMS TO VALUE**. В *ProgramName2* можно указать файл, в котором размещается вызываемая программа.

В **Visual Basic** сходные задачи выполняет команда

Call Name([ParameterList])

Как ее использовать покажем на примере вызова на выполнение подпрограммы:

```
Call PrintToDebugWindow("Печать в окне отладки")
Sub PrintToDebugWindow(cString)
Debug.Print AnyString
End Sub
```

8. Ранее мы научились определять в программе нужные нам переменные с требуемым типом данных. При обработке данных программисту в силу различных причин часто приходится преобразовывать данные из одного типа в другой.

В языке программирования **Visual FoxPro** для этого существует большое количество функций, из которых наиболее часто используемыми можно назвать следующие:

STR(*nExpression* [, *nLength* [, *nDecimalPlaces*]])

Преобразует числовое выражение *nExpression* в строку символов. Необязательные параметры *nLength* и *nDecimalPlaces* позволяют задать длину и число десятичных разрядов соответственно.

VAL(*cExpression*)

Преобразует в число строку символов, представляющую собой набор цифр.

В **Visual Basic** существуют аналогичные функции:

Str(*nExpression*)

Val(*cExpression*)

7. Трудно представить себе программу обработки данных, в которой программисту не пришлось бы с этими данными хоть что-то делать. С числами все просто. В этой области мы постоянно тренируем себя при посещении магазинов. Суммы растут, тренировки становятся все более напряженными. Сложнее с символьными данными. И здесь мы должны включить в почетную десятку хотя бы основные функции для работы со строками.

В **Visual FoxPro** по частоте употребления можно выделить следующие функции:

SUBSTR(*cExpression*, *nStartPosition* [, *nCharactersReturned*])

Возвращает фрагмент строки символов из строкового выражения *cExpression*, начинающийся с позиции *nStartPosition* и длиной *nCharactersReturned*.

LEFT(*cExpression*, *nExpression*)

Возвращает из строкового выражения *cExpression* фрагмент, включающий первые *nExpression* символов.

RIGHT(*cExpression*, *nExpression*)

Возвращает из строкового выражения *cExpression* фрагмент, включающий последние *nExpression* символов.

ALLTRIM(*cExpression*)

Исключает все начальные и конечные пробелы из строкового выражения.

В **Visual Basic** существуют аналогичные функции:

Mid(*cExpression*, *nStartPosition* [, *nCharactersReturned*])
 Left(*cExpression*, *nExpression*)
 Right(*cExpression*, *nExpression*)
 Trim(*cExpression*)

6. При работе с данными нам постоянно приходится думать об их "свежести", поэтому мы никак не можем обойти вниманием функции, связанные с определением текущей даты и времени.

В Visual FoxPro для этого можно использовать следующие функции:

Для определения текущей даты

DATE()

Для определения текущего времени

TIME([*nExpression*])

Для определения текущей даты и времени в формате дата и время

DATETIME()

Если в функции TIME() в качестве выражения задать любую числовую величину, значение текущего времени будет возвращено с сотыми долями секунд, хотя тогда лучше уж обратиться к функции SECONDS().

В Visual Basic существуют аналогичные функции:

Date
 Time
 Now

5. В том случае, если необходимо многократно выполнять какой-либо блок команд, пока выполняется заданное условие, может быть использована команда, которая в Visual FoxPro выглядит вот так

DO WHILE *lExpression Commands* [LOOP] [EXIT]
 ENDDO

Каждый раз, когда программа достигает строки с командой **DO WHILE**, она проверяет значение выражения *lExpression*, и если оно равно .T., то выполняются команды внутри структуры, если .F., то управление передается строке, следующей за ENDDO. Опция LOOP позволяет после ее указания вернуть управление к строке DO WHILE, а опция EXIT - прекратить выполнение цикла, невзирая на значение *lExpression*.

Наиболее часто в качестве условия используют выражения:

- **DO WHILE .NOT. EOF()** - до исчерпания записей в таблице;
- **DO WHILE nMin < 10 .AND. nMax > 100** - пока переменная nMin меньше 10 и переменная nMax больше 100;
- **DO WHILE .T.** - бесконечно выполняемый цикл (выход только по EXIT).

Естественно, что в случае использования опций LOOP и (или) EXIT перед ними должны быть записаны свои условия их выполнения (чаще всего команда **IF...ENDIF**). Приведем простой пример организации цикла для реакции на действия пользователя.

```
DO WHILE .T.
  CLEAR
  WAIT "Наберите цифру 1 или 2" TO cNumber
  DO CASE
    CASE cNumber = "1"
      ? "Вы набрали цифру 1"
    CASE cNumber = "2"
      ? "Вы набрали цифру 2"
    OTHERWISE
      ? "Ошибка: введенное значение не 1 или 2!"
  ENDCASE
  WAIT "Хотите попробовать еще раз (Y/N)" TO cYesNo
  IF UPPER(cYesNo) <>> "Y"
    EXIT
  ENDIF
ENDDO
```

В этом примере для обеспечения ввода пользователем данных используется простейший вариант команды **WAIT**, которая приостанавливает работу программы и после нажатия пользователем какой-либо клавиши присваивает значение этой клавиши переменной, указанной в опции **TO**.

В Visual Basic для записи этой команды надо использовать один из следующих эквивалентных вариантов синтаксиса:

```
Do [{While | Until} Expression] [Commands] [Exit Do] [Commands]  
Loop
```

или

```
Do [Commands] [Exit Do] [Commands]  
Loop [{While | Until} Expression]
```

Как правильно использовать эту команду, видно из следующего примера:

```
ICheck = True  
nCounter = 0  
Do ` Первый (внешний) цикл  
Do While nCounter < 20 ` Второй (внутренний) цикл  
    nCounter = nCounter + 1  
    If nCounter = 10 Then  
        ICheck = False  
        Exit Do ` Выход из второго цикла  
    End If  
Loop  
Loop Until ICheck = False ` Выход из первого цикла
```

4. Если мы запишем в программу некую последовательность команд, то они будут выполняться построчно от начала до конца программы. Вряд ли таким образом мы сможем написать мощную программу, которая должна обрабатывать действия пользователя в зависимости от тех или иных условий. Нам потребуются средства для управления последовательностью выполнения команд.

Наиболее простое ветвление программы в зависимости от заданного условия можно обеспечить командой, синтаксис которой в Visual FoxPro записывается в следующем виде:

```
IF !Expression  
    Commands  
[ELSE  
    Commands]  
ENDIF
```

В качестве условия может быть использовано любое выражение, в том числе и логическое. На месте команд, в свою очередь, может стоять снова команда **IF** (вложенное условие). Если выражение равно **.T.**, то выполняются команды, заключенные между **IF** и **ELSE**. Если **.F.**, то выполняются команды, следующие за **ELSE**, а в том случае, если эта необязательная опция **ELSE** в команде не используется, выполняются команды, следующие за **ENDIF**.

В Visual Basic эту команду можно записать на одной строке:

```
If !Expression Then Commands [Else Commands]
```

или в более сложном виде:

```
If Expression Then  
    [Commands]  
[ElseIf Expression_n Then  
    [Commands]] ...  
[Else  
    [Commands]]  
End If
```

Если выражение *Expression* имеет значение **True**, выполняются команды, следующие за ключевым словом **THEN**. В противном случае выполняется один из блоков, в котором

Expression_n будет соответствовать выражению *Expression*. Если и этого не произойдет, будет выполнен блок команд, следующих за ключевым словом ELSE.

Для выбора по условию очень эффективно можно использовать функцию, которая по смыслу похожа на рассматриваемую команду и имеет совершенно одинаковый синтаксис как в Visual FoxPro, так и в Visual Basic

IIF(*IExpression*, *Expression1*, *Expression2*)

Возвращает значение *Expression1*, если *IExpression* = .T., и значение *Expression2*, если *IExpression* = .F.; *Expression1* и *Expression2* должны быть одного типа, например:

TITLE = IIF(nSign=0,"автомобиль","автопоезд") + cModel

Эта функция может использоваться и для выбора выполняемых подпрограмм, например:

```
cProc_var = IIF(nVid>>5,cProc_1,IIF(nVid<<2,cProc_2,cProc_3))
DO (cProc_var)
```

Здесь, в зависимости от значения переменной *nVid*, будет выполняться та или другая процедура, причем для увеличения числа вариантов в качестве *Expression2* используется еще одна - вложенная функция **IIF()**. Использование этой функции для программного ветвления, там где это возможно, значительно увеличивает быстродействие программы и делает ее значительно меньше, чем при употреблении команды **IF...ENDIF**.

3. Более сложное ветвление, когда есть несколько условий, в зависимости от которых надо выполнить ту или иную группу команд, организуется структурной командой, которая в Visual FoxPro выглядит так:

```
DO CASE
  CASE IExpression1
    Commands
  [CASE IExpression2
    Commands]
  ...
  [OTHERWISE
    Commands]
ENDCASE
```

Команда выполняет проверку условий, заданных в операторах CASE. Они просматриваются последовательно сверху вниз, и, как только первое из них оказывается верным (логическое выражение равно .T.), выполняется блок команд для этого CASE, после чего управление передается строке программы, следующей за ENDCASE. Если ни одно из условий не истинно, выполняется опция OTHERWISE, при отсутствии этой опции команда игнорируется.

Например:

```
nChoice = 3
DO CASE
  CASE nChoice = 1
    ? "Выбрано значение 1"
  CASE nChoice = 2
    ? "Выбрано значение 2"
  OTHERWISE
    ? "Выбрано неправильное значение!"
  RETURN
ENDCASE
```

В Visual Basic синтаксис этой структурной команды организован несколько по-иному:

```
Select Case Expression
[Case Expression_n
  [Commands]] ...
[Case Else
  [Commands]]
End Select
```

Выполняются те блоки CASE, в которых выражение *Expression_n* соответствует выражению *Expression*. Если такого соответствия не обнаруживается, то выполняется блок CASE ELSE.

Приведенный выше пример можно записать примерно так:

```

Dim nChoice As Integer
Select Case nChoice
    Case 1
        Debug.Print "Выбрано значение 1"
    Case 2
        Debug.Print "Выбрано значение 2"
    Case Else
        Debug.Print "Выбрано неправильное значение"
    End
End Select

```

2. Цикл, помимо структуры **DO WHILE**, можно организовать и стандартным для большинства языков программирования способом.
Поэтому следующая команда куда популярнее предыдущей:

```

FOR MemVarName = nInitialValue TO nFinalValue [STEP nIncrement]
    Commands
[LOOP]
[EXIT]
NEXT

```

Команда **FOR...NEXT** обеспечивает выполнение блока команд для каждого значения переменной, начиная со значения, равного *nInitialValue*, до значения *nFinalValue*, пошаговое увеличение или уменьшение происходит на величину, заданную *nIncrement*, которая по умолчанию равна 1. Опция **LOOP** приводит к передаче управления на начало следующего цикла, а **EXIT** - к прекращению действия

команды.

В Visual Basic синтаксис этой команды выглядит очень похоже:

```

For MemVarName = nInitialValue To nFinalValue [Step nIncrement]
    [Commands]
[Exit For]
[Commands]
Next [MemVarName]

```

По сравнению с Visual FoxPro здесь нет возможности "досрочно" начать новый цикл с помощью опции **LOOP**.

Циклы **FOR...NEXT** могут быть вложены, но в этом случае в каждом цикле должен использоваться уникальный параметр *MemVarName*. В синтаксисе Visual FoxPro пример организации вложенного цикла приведен ниже. В переменную *MyString* десять раз записываются значения от 0 до 9.

```

MyString = " "
FOR Words = 10 TO 1 Step -1
    FOR Chars = 0 To 9
        MyString = MyString + STR(Chars)
    NEXT
    MyString = MyString + " "
NEXT
? MyString

```

В синтаксисе Visual Basic этот же пример будет иметь лишь незначительные отличия:

```

Public Sub Main()
For Words = 10 To 1 Step -1
    For Chars = 0 To 9
        MyString = MyString & Chars
    Next
    MyString = MyString & " "
Next

```

```
Debug.Print MyString
End Sub
```

В Visual Basic есть очень похожая на рассматриваемую команду команда организации цикла:

```
For Each MemVarName In ArrayName
[Commands]
[Exit For]
[Commands]
Next [MemVarName]
```

Эта команда позволяет выполнить какие-либо действия для группы элементов массива, имя которого указывается в параметре *ArrayName*. Вместо массива может использоваться коллекция объектов, о чем мы будем более подробно говорить позднее.

1. Даже если мы научились составлять пока очень простенькие программы, мы должны стремиться к высоким идеалам и стараться создать дружелюбный интерфейс. Здесь не обойтись без средств, позволяющих в хорошей манере сообщать пользователям разную информацию, может и не очень приятную, спрашивать у них совета о дальнейших действиях и т. д. Учитывая важность этой проблемы для современного пользовательского приложения, авторы, заранее сговорившись, на первое место поставили следующую функцию. В Visual FoxPro она имеет такой синтаксис:

MESSAGEBOX(*cMessageText* [, *nDialogBoxType* [, *cTitleBarText*]])

Выводит на экран определяемое пользователем диалоговое окно. Параметр *cMessageText* определяет текст, который появляется в этом окне. Чтобы переместить часть сообщения на следующую строку в диалоговом окне, используйте в *cMessageText* символ возврата каретки (CHR(13)). Высота и ширина диалогового окна увеличиваются по мере необходимости, чтобы вместить содержимое заданного в *cMessageText* текста.

Параметр *nDialogBoxType* определяет кнопки и пиктограммы, которые появляются в диалоговом окне, а также первоначально выбранную после вывода диалогового окна на экран кнопку.

В следующей таблице приведены возможные значения для этих элементов.

Значение <i>nDialogBoxType</i>	Элементы
	Кнопки диалогового окна
0	Только кнопка OK
1	Кнопки OK и Cancel
2	Кнопки Abort, Retry и Ignore
3	Кнопки Yes, No и Cancel
4	Кнопки Yes и No
5	Кнопки Retry и Cancel
	Пиктограмма
16	Знак "Стоп"
32	Знак "Вопрос"
48	Знак "Восклицание"
64	Пиктограмма "Информация (i)"
	Кнопка по умолчанию
0	Первая кнопка
256	Вторая кнопка
512	Третья кнопка

Пропуск значения *nDialogBoxType* идентичен определению для него значения 0.

Параметр *nDialogBoxType* должен представлять собой сумму трех значений - по одному из каждого раздела приведенной таблицы.

Параметр *cTitleBarText* определяет текст, который появляется в заголовке диалогового окна. Если вы опускаете *cTitleBarText*, в названии окна появится заглавие "Microsoft Visual FoxPro".

Возвращаемое значение функции MESSAGEBOX() указывает, какая кнопка была выбрана в

диалоговом окне. В диалоговых окнах с кнопкой Cancel при нажатии клавиши Esc будет возвращаться то же самое значение (2), что и при выборе Cancel.

Следующая таблица содержит список возвращаемых функцией MESSAGEBOX() значений.

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

В Visual Basic рассматриваемая функция имеет чуть-чуть другое написание и несколько более богатый синтаксис:

`MsgBox(cMessageText [, nDialogBoxType [, cTitleBarText] [, cHelpFileName, nContext])`

Условия использования этой функции и задания ее параметров идентичны описанным выше для Visual FoxPro. В Visual Basic для некоторых параметров можно устанавливать дополнительные значения. Так, параметр *nDialogBoxType* дополнительно может принимать значение 4096, что устанавливает модальность появляющегося сообщения на системном уровне, и пользователь не может продолжить работать не только в текущем, но и ни в каком другом из запущенных приложений, пока не отреагирует на данное сообщение.

Параметры *cHelpFileName* и *nContext* позволяют задать имя файла контекстной помощи и номер темы, которая будет выведена на экран при нажатии пользователем клавиши F1.

4.4. Еще несколько навязчивых советов

Когда вы напишете пусть даже небольшую программу, посмотрите, сколько имен вы использовали. А если программа достаточно сложная? Что бы не запутаться в громадном количестве имен, используемых в программе, программисты давно выработали негласные соглашения, с помощью которых здесь можно навести хоть какой-то порядок. Надо отметить, что, начиная с версии 3.0, такие рекомендации включены и в руководство по Visual FoxPro и по Visual Basic. Их основная идея заключается в систематизации всех имен переменных или массивов по диапазону действия и типу хранимых в них данных.

Таким образом, структура имени должна иметь следующий вид:
ДиапазонТип_данныхИмя

Для указания диапазона рекомендуется использовать следующие символы:

l	- внутренняя (local)	lnCounter
p	- локальная (private)	pnStatus
r	- региональная (region)	rnCounter
g	- глобальная (public)	gnOldRecno
t	- параметр (parameter)	tnRecNo

В последнем столбце приведен пример наименования. Надо отметить, что в имени переменной диапазон ее действия программисты указывают редко. В примерах, где это не имеет принципиального значения, мы тоже не будем делать этого.

Для указания типа данных рекомендуется использовать следующие символы:

a	- массив	aMonths
c	- символьные данные	cLastName
y	- денежная единица	yCurrentValue
d	- дата	dBirthDay
t	- данные типа "дата и время"	tLastModified
b	- двойной точности	bValue
l	- логические	lFlag
n	- числовые	nCounter
o	- ссылка на объект	oEmployee

u - неопределенного типа uReturnValue

Как вы могли заметить из примеров, после обозначения диапазона действия и типа данных, имя переменной для большей наглядности лучше начинать с большой буквы. Не ленитесь давать имени максимальную смысловую нагрузку. Это избавит вас от долгих раздумий: "И что же это за переменная такая и зачем это она тут?"

Глава 5

Объектно-ориентированное программирование

5.1. Объектная модель и ее свойства

5.2. Объекты и их свойства

Объекты для работы с данными
Объекты для управления работой приложения
Объекты для оформления интерфейса пользователя
Объекты-контейнеры
Невизуальные объекты
Объекты OLE

5.3. Управление событиями

5.4. Использование методов

Программирование - не такое уж приятное занятие, как может показаться на первый взгляд. Регулярно программисты сталкиваются с двумя сильно действующими на нервы событиями: они то часами мучаются в раздумьях, как запрограммировать то или иное функциональное решение, то их одолевает ужасная скука при многократной реализации давно отработанных решений. Способ борьбы с этими неприятностями называется "объектно-ориентированное программирование" (ООП).

По разным причинам ООП, как неотъемлемая часть современных систем RAD, не сразу стала достоянием широкого круга программистов, долгое время оставаясь уделом профессионалов, пишущих на языках, подобных C++ . Среди рассматриваемых в данной книге средств разработки наиболее развитые средства ООП имеет СУБД Visual FoxPro. Нисколько не сомневаясь, что остальные средства разработки будут двигаться в этом направлении, при изложении основ ООП мы будем ориентироваться именно на эту модель.

5.1. Объектная модель и ее свойства

При структурном программировании, речь о котором шла в [предыдущей главе](#), последовательно выполняются операторы, записываемые программистом в соответствии с логикой решения поставленной перед ним задачи. А теперь представьте интерфейс современной прикладной программы, что-нибудь типа Windows 95. Представили это многообразие окон, диалогов, задач? И все это то открывается, то закрывается, то перемещается, то изменяет цвет. Рискнем предположить, что такой интерфейс написать средствами стандартного структурного языка программирования просто невозможно.

В этом параграфе вы узнаете:

- Основные сведения о принципах ООП.
- Термины и определения, используемые в ООП.
- Возможности объектной модели Visual FoxPro.
- Особенности использования объектов в Visual Basic.

Объектно-ориентированное программирование делает акцент не на программные структуры, а на объекты. Почти все, что представляет для вас интерес и, возможно, даже не существует визуально, может быть объектом. Объектом может быть окно, поле для ввода данных, пользователь вашей программы, сама программа и т. д. Тогда любые действия мы можем привязать к этому объекту, а также описать, что должно случиться с ним при определенных действиях пользователя (например, при закрытии окна). Полезный, многократно используемый

объект можно сохранить в качестве типового и применять его в нескольких программах для обеспечения сходной функциональности.

Таким образом, в ООП программист создает нужные объекты, а затем описывает действия с ними и их реакцию на действия пользователя. При таком подходе любую программу можно написать, включая туда тот или иной набор объектов, обеспечивающих выполнение тех или иных функций.

Более детальное знакомство с принципами ООП начнем с терминологии, т. к. некоторые понятия здесь являются не только новыми, но и не такими простыми, как может показаться на первый взгляд.

Абстрактным типом данных называется такой тип данных, который описан программистом, но не поддерживается в используемом языке программирования.

В языках программирования, которые не поддерживают ООП, таких как, например, стандартный С, абстрактный тип данных может быть создан из существующих типов данных. Хорошим примером такого подхода является использование структур. В языках, поддерживающих ООП, на понятие "абстрактный тип данных" основано описание классов, осуществляемое в программе на высоком уровне. В процессе работы программы это описание используется для создания объектов, выполняющих реальные функции. В Visual FoxPro такой тип данных поддерживается программно за счет использования команды **DEFINE CLASS** и визуально при использовании Конструктора классов (Class Designer). Объекты, созданные во время работы программы, доступны для управления посредством специальных переменных, которые служат в качестве указателей на эти объекты.

Абстракцией называется метод, при использовании которого можно игнорировать не существенные в данный момент детали и функции программы для того, чтобы получить возможность сосредоточиться на интерфейсе программы, обмене сообщениями с пользователем и т. д.

Абстракция поддерживает идею "черного ящика", когда часть или даже вся информация о поведении объекта может быть скрыта. Такой подход обеспечивает возможность использования в пользовательском приложении объектов с заранее определенными свойствами, функциональность которых не может опускаться ниже определенного уровня. В то же время возможно придание таким объектам дополнительных свойств и функций, изменение их внешнего вида и т. д.

Классом называется шаблон, который описывает методы и свойства используемые для определенного типа объектов.

На основании описания класса при работе программы создаются объекты, которые и обеспечивают выполнение заданных функций. Из описания одного класса может быть создано сколько угодно объектов. Такие объекты будут называться экземплярами одного класса. Описать класс программно можно с помощью команды **DEFINE CLASS** и визуально при использовании Конструктора классов.

В Visual FoxPro классы делятся на визуальные и не визуальные. **Визуальные классы** служат прообразами объектов, которые будут видны и, соответственно, станут основой пользовательского интерфейса будущей программы. **Не визуальные классы** могут быть видны только в момент проектирования на их основе объектов, которые окажутся невидимыми при работе программы. Как правило, объекты, основанные на не визуальных классах, создаются и управляются программно, с помощью соответствующих команд и функций.

Классы хранятся в файле **библиотеки классов**, что облегчает к ним доступ. Этот файл имеет расширение **VCX** и может содержать один или несколько классов.

Объектом называется программно связанная коллекция методов (функций) и свойств, выполняющих одну функционально связанную задачу.

Например, объект - окно для вывода данных - обладает набором свойств, описывающих его внешний вид, и набором методов для управления его поведением на экране. Программно объект может быть создан с помощью функции **CREATEOBJECT()**.

Свойство - это характеристика, с помощью которой описывается внешний вид и работа объекта. Например, заголовок, тип шрифта для его написания, вид линий и цвет в обрамляющей рамке, источник данных и т. д.

Событие - это действие, которое связано с объектом. Событие может быть инициировано пользователем, вызвано программой или операционной системой.

События являются основным инструментом для описания требуемой реакции объекта на действия пользователя. Все возможные события содержат по умолчанию заложенную в них реакцию, которую программист может изменить, записав в процедуру события определенный программный код. Эти процедуры могут быть защищены от дальнейших изменений или, наоборот, легко доступны для модификации пользователем. Некоторые действия в программе вызывают целую череду событий, происходящих в определенной последовательности. Например, загрузка экранной формы вызывает события, последовательность которых не может быть изменена.

Метод - это функция или процедура, которая будет управлять работой объекта. Момент выполнения метода определяется исключительно программистом, т. к. метод вызывается только при его явном указании, например **OBJECT.DRAW**.

Метод может быть защищен от дальнейшего изменения. Как в любую функцию, в него можно передать параметры и получить возвращаемое значение. Однако, ни метод, ни событие не могут иметь вложенных процедур. Стоит также учесть, что метод не может носить имя, совпадающее по названию со свойством.

Иерархия классов - это древообразная структура классов, отображающая взаимосвязи между используемыми в приложении классами одного типа.

Разобраться в понятии "иерархия классов" вам поможет рис. 5.1. Классы, находящиеся на более низких ступенях иерархии, называются *подклассами*. Любой из классов, находящихся в иерархии на рис. 5.1 ниже базового класса, будет подклассом. Классы, находящиеся на более высоких ступенях, на основании которых описаны ниже стоящие в иерархии классы, называются *родительскими классами* или *суперклассами*. Например, на рис. 5.1 базовый класс разработчика будет родительским по отношению к классу кнопок для панели инструментов. На основании одного родительского класса может быть создано сколько угодно подклассов следующего уровня. Такие подклассы называются экземплярами данного класса. На рис. 5.1 такими экземплярами будут классы для выполнения определенных действий, созданные на основе родительского класса графических кнопок.

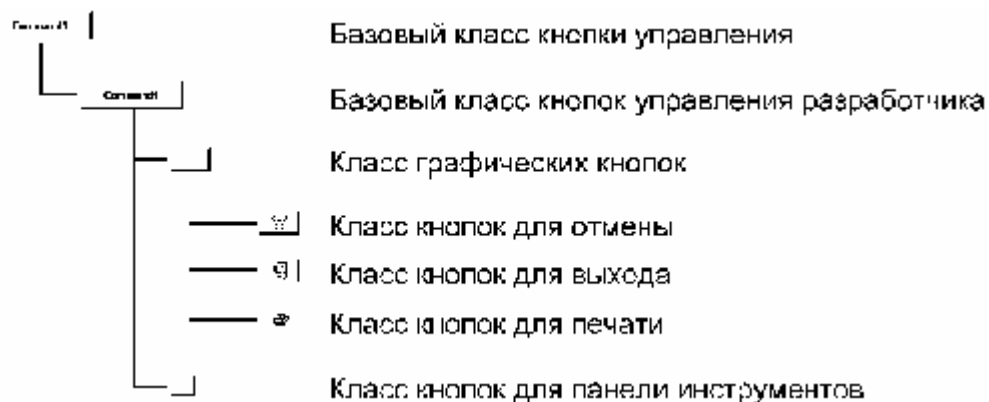


Рис. 5.1.

Иерархия классов может включать столько уровней, сколько пожелает разработчик, однако при разработке системы классов для приложения лучше не превышать пяти уровней иерархии.

Базовым классом называется класс, находящийся на вершине иерархии классов, используемых в приложении.

В Visual FoxPro базовые классы имеют весьма серьезную особенность. Они встроены в саму СУБД, и, следовательно, их описание не может быть изменено. В связи с этим полезно ввести понятие "базовых классов разработчика". **Базовые классы разработчика** являются дублерами базовых классов Visual FoxPro, стоят на следующей ступени иерархии после них, но являются классами более высокого уровня для всех остальных классов, используемых в приложении, как это видно на рис. 5.1 на примере классов для кнопок управления.

Всего в Visual FoxPro программист может использовать около 30 базовых классов. Их классификация приведена на рис. 5.2, а назначение описано в табл. 5.1.

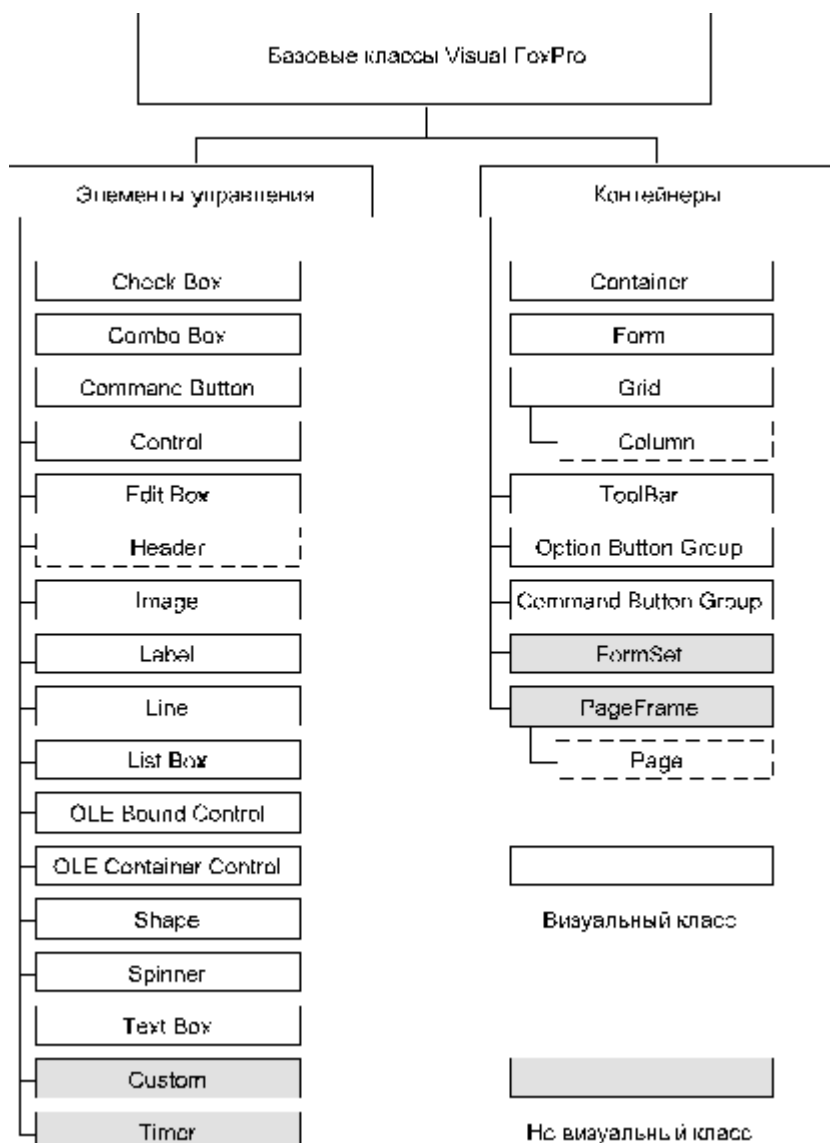


Рис. 5.2.

Таблица 5.1. Базовые классы Visual FoxPro

Имя класса	Описание
CheckBox	Создает поле проверки, которое используется для переключения между двумя состояниями.
Column	Создает столбец в объекте Grid. Столбец может содержать данные из поля в таблице

	или выражение, а также включать в себя какие-либо другие элементы управления.
ComboBox	Создает список, из которого можно выбрать один элемент. Сочетает возможности элементов управления ListBox и TextBox , так как, помимо выбора из списка, позволяет вводить данные.
CommandButton	Создает одиночную кнопку управления. Кнопка обычно используется, чтобы активизировать событие, подобное закрытию формы, перемещению курсора в другую запись, печати отчета и т. д.
CommandGroup	Создает группу кнопок управления.
Container	Создает объект, который может содержать другие объекты. Объекты Container могут содержать другие объекты и разрешают доступ к объектам, содержащимся внутри них.
Control	Создает объект элемента управления, который может содержать другие защищенные объекты. Объекты Control могут включать в себя другие объекты, но в отличие от объектов Container не позволяют осуществлять доступ к объектам, находящимся внутри них
Custom	Создает определяемый пользователем объект на основе пользовательского класса. Определяемые пользователем классы - это классы со свойствами, событиями и методами, но без визуального представления.
EditBox	Создает область редактирования. Используйте элемент управления EditBox для символьных полей большой длины или полей примечаний.
Form	Создает форму для работы с данными и управления работой программы. Форма - это объект-контейнер, который включает в себя все необходимые элементы управления и составляет основу пользовательского интерфейса.
FormSet	Создает объект-контейнер, который содержит набор форм.
Grid	Создает объект Grid . Grid - это объект-контейнер, отображающий данные в строках и столбцах и по внешнему виду похожий на окно Browse , но имеющий расширенную функциональность, так как вы имеете полный контроль над каждым элементом в Grid за счет отдельного набора свойств.
Header	Создает заголовок для столбца в Grid . Объект Header позволяет отвечать на события, то есть может изменять свое значение в процессе работы программы.
Image	Создает элемент управления, который показывает изображение в формате BMP .
Label	Создает метку, которая отображает текст.
Line	Создает элемент управления, отображающий горизонтальную, вертикальную или диагональную линию.
ListBox	Создает поле списка. Этот элемент отображает список пунктов, из которых вы

	<p>можете выбрать один или несколько. Может применяться при вводе данных, когда пользователь должен ввести только заранее определенные значения.</p>
OLE Bound Control	<p>Создает связанный элемент управления OLE. Связанный элемент управления OLE позволяет вам добавлять включаемые объекты OLE из других прикладных программ типа Microsoft Word и Microsoft Excel, поддерживающих стандарт OLE 2.0.</p>
OLE Container Control	<p>Создает элемент управления OLE. OLE-объекты содержат элементы управления ActiveX (файлы с расширением OCX) и включаемые OLE-объекты из других прикладных программ типа Microsoft Word и Microsoft Excel. В отличие от элементов управления ActiveX, включаемые OLE-объекты не имеют собственного набора событий. Кроме того, элементы управления ActiveX не привязаны к полю типа General в таблице FoxPro, как связанные элементы управления OLE.</p>
OptionButton	<p>Создает одиночную кнопку выбора. Такая кнопка может быть добавлена только к группе кнопок выбора.</p>
OptionGroup	<p>Создает группу кнопок выбора. Группа кнопок выбора - это контейнер, который содержит отдельные кнопки выбора. Позволяет пользователю выбрать одно действие из списка возможных вариантов, представленных набором кнопок.</p>
Page	<p>Создает страницу в страничном блоке. Объект Page позволяет с легкостью создавать многостраничные формы или диалоги путем помещения в окно формы набора страниц, переключаемых с помощью вкладок.</p>
PageFrame	<p>Создает страничный блок, в котором содержатся страницы формы. Страничный блок определяет глобальные характеристики страницы формы: размер и положение на экране, вид рамки, активную страницу и т. д.</p>
Shape	<p>Создает элемент управления формы, который отображает геометрическую фигуру (прямоугольник, круг или эллипс).</p>
Spinner	<p>Создает счетчик для фиксированного изменения числового значения.</p>
TextBox	<p>Создает текстовое поле, в котором можно редактировать содержимое переменной, элемента массива или поля. Текстовое поле - это один из самых широко используемых элементов управления для ввода и редактирования заранее не определенных величин.</p>
Timer	<p>Создает невидимый во время работы программы объект, который обеспечивает контроль по времени за выполнением всей программы или отдельных ее фрагментов. Управление с помощью таймера является полезным для фоновой обработки. Типичное использование таймера - это получение системного времени для определения</p>

ToolBar

момента, когда необходимо выполнить какую-либо программу или прервать некоторое действие.

Создает пользовательскую панель инструментов. Панель инструментов представляет собой набор объектов, которые объединены в одном окне. Панель инструментов может быть пристыкована к верхней, нижней или боковым рамкам главного окна. Если панель не пристыкована, она ведет себя аналогично форме.

Базовые классы делятся на контейнеры и элементы управления. Стоит пояснить несколько подробнее отличие классов, на основе которых создаются будущие объекты - элементы управления, от классов-контейнеров. **Объекты-контейнеры** могут содержать внутри себя другие объекты, в то же время допуская манипуляции с этими внутренними объектами. Их можно назвать составными объектами, при этом отдельные составляющие части не теряют своего "суверенитета". Объекты - **элементы управления**, основанные на не контейнерных классах, хотя тоже могут состоять из нескольких составных частей, допускают манипуляции с ними только как с единым компонентом.

Некоторые базовые классы занимают особое положение в связи с тем, что сами являются составной частью другого базового класса и поэтому не могут быть основой для создания подкласса (на рис 5.2 выделены пунктирной линией).

Наследованием называется способность передачи свойств и методов, принадлежащих классу, на основе которого создается подкласс, вновь создаваемому классу.

Вновь создаваемый подкласс автоматически наследует все свойства и методы родительского класса, но вы всегда можете изменить какие-то из этих свойств или методов для выполнения специализации данного подкласса. Наследование поддерживается не только при создании подкласса, но и в дальнейшем. Таким образом, все сделанные вами изменения в родительском классе тут же отразятся на его подклассах. Как видно на рис. 5.1, класс кнопок для вывода данных на печать наследует свой внешний вид от класса графических кнопок, но имеет специализированное изображение и, очевидно, специфическую реакцию на нажатие. Если мы решим изменить размер кнопки в классе графических кнопок, изменятся размеры всех используемых в приложении кнопок, стоящих ниже в иерархии классов.

Оператор указания диапазона позволяет вызвать метод родительского класса с более низкого уровня, в пределах описания подкласса. Это позволяет расширить функциональность объекта без необходимости написания лишнего программного кода. При создании подкласса он автоматически наследует все методы родительского класса. Мы можем изменить унаследованный метод и в то же время выполнить не только этот измененный метод для данного подкласса, но и метод родительского класса. Оператор имеет следующий синтаксис:

cClassName::Method

Инкапсуляция - это возможность объединения связанных фрагментов данных или процессов в отдельный модуль - контейнер.

Это похоже на понятие абстракция - скрытие внутренних данных, то есть использование принципа создания объекта как "черного ящика". Такой объект будет работать без раскрытия своей внутренней структуры, обеспечивающей его функциональность. Элементы управления, основанные на не контейнерных классах, являются хорошим примером использования принципа инкапсуляции. Использование инкапсуляции дает два очевидных преимущества программисту:

- Более простой процесс разработки программы. При создании объектов программист может сосредоточиться на более узких, конкретных задачах без необходимости обдумывания тех последствий, которые могут произойти в других частях программы из-за сделанных им изменений.
- Более безопасный способ дублирования фрагментов кода или объектов. После того как объект описан и правильность его работы в программе проверена, все программы или

части системы, работающие по сходному алгоритму, например доступа к данным, легко создаются как порожденные объекты. Это исключает риск повреждения данных при независимом программировании каждой части системы и возможном при этом появлении большого числа ошибок.

Полиморфизм - это возможность осуществлять одинаковое обращение к различным объектам в случае, если каждый объект может выполнять такое обращение.

Полиморфизм обеспечивает общий интерфейс для работы с создаваемыми объектами. Например, вызов метода **Draw** для объектов **Box1** и **Box2** может вызвать совершенно различные последствия, так как каждый из этих объектов может иметь собственный метод **Draw**. Для программиста полиморфизм обеспечивает более простое и гибкое управление для группы связанных объектов.

Обращение - это инструкция от одного объекта к другому для выполнения одного из методов того объекта, к которому обращаются.

Различают три части обращения: имя объекта, получающего сообщение, имя метода и, возможно, передаваемые параметры, если того требует выполняемый метод.

В обращении имя объекта отделяется специальным оператором - точкой (.). Например, если надо сделать недоступным объект - список с именем **MyList**, необходимо выполнить команду: **MyList.Enabled = .F.**

Это вызовет изменение свойства, контролирующего доступность данного объекта для работы с ним пользователя. Список изменит свой внешний вид и для возвращения его в рабочее состояние нужна команда:

MyList.Enabled = .T.

Обращение к объекту для выполнения каких-либо действий аналогично обращению для изменения его свойств. После имени объекта мы должны указать имя метода:

MyList.Refresh()

В некоторых ситуациях мы не сможем прямо обратиться к интересующему нас объекту. Например, если мы захотим обратиться к упомянутому выше списку из другой формы, то должны будем послать наше обращение не списку, а форме, содержащей нужный список:

MyForm.MyList.Enabled = .T.

Такой порядок обращения к объекту может показаться весьма утомительным, особенно если представить себе достаточно длинную цепочку вложенных объектов. Однако это позволяет не подыскивать сотню уникальных имен для кнопки выхода в каждой из 100 используемых в приложении форм. Все кнопки могут иметь одно и то же имя **cmdClose**, и для каждого обращения найдется правильный адресат, т. к. в них будет использоваться имя нужной формы:

FormName.cmdClose

Значительно сократить программный код позволяют операторы относительной ссылки, в которых, в отличие от абсолютной ссылки, не используются имена объектов:

- **THIS** - позволяет сослаться на текущий объект;
- **THISFORM** - позволяет сослаться на текущую форму;
- **THISFORMSET** - позволяет сослаться на текущий набор форм.

В командах эти операторы могут указываться вместо соответствующих имен объектов, форм или наборов форм при выполнении какого-либо метода или изменении значения свойств. В каких случаях какие операторы относительной ссылки надо использовать, поможет разобраться табл. 5.2.

Таблица 5.2. Обращение к объектам с помощью относительной адресации

Источник обращения	Пример обращения к объекту MyObject
Другой метод того же самого объекта	THIS.Enabled = .T.
Форма, включающая объект	THIS.MyObject.Enabled = .T.

Другой объект той же самой формы	<code>THISFORM.MyObject.Enabled = .T.</code>
Извне формы	<code>MyForm.MyObject.Enabled = .T.</code>
Объект другой формы, входящей в один набор форм	<code>THISFORMSET.MyForm.MyObject.Enabled = .T.</code>
Извне набора форм	<code>MyFormSet.MyForm.MyObject.Enabled = .T.</code>

В Visual FoxPro и Visual Basic есть два свойства, которые позволяют выполнять действия относительно активного объекта, то есть объекта, с которым в данный момент работает пользователь (на который указывает курсор), без необходимости знания его имени. Свойство **ActiveForm** формы или набора форм имеет следующий синтаксис:

`FormSet.ActiveForm.Property [= Setting]`

или

`FormSet.ActiveForm.Method`

Это свойство позволяет узнать заданное в *Property* свойство для активной формы или выполнить указанный в *Method* метод. Если мы задаем параметр *Setting*, то указанное значение сравнивается с установленным, и в случае равенства возвращается .T. Если *Setting* не указывается, возвращается установленное значение свойства *Property*.

Если мы не используем набор форм, то на месте **FormSet** в Visual FoxPro мы должны сослаться на экранный объект с помощью системной переменной **_SCREEN**. Эта системная переменная выполняет в таком случае несколько нестандартные для системных переменных в Visual FoxPro функции. Она играет роль ссылки на главное окно Visual FoxPro, позволяя управлять им как объектом. Например, для очистки главного окна Visual FoxPro от выведенных данных можно в окне *Command* написать команду **CLEAR**. Тот же самый результат будет достигнут, если мы напишем следующую строку:

`_SCREEN.Cls`

В Visual Basic с этой же целью используется объект **Screen**. Например:

`Screen.ActiveForm.MousePointer = 4`

Свойство **ActiveControl** позволяет сослаться на активный объект формы, страницы в многостраничной форме или панели инструментов. Синтаксис этого свойства следующий:
`Object.ActiveControl.Property [= Setting]`

Если во время работы с формой мы в окне *Debug* наберем приведенную ниже строчку и будем перемещаться между разными элементами управления, то увидим имя активного в данный момент элемента управления:

`_SCREEN.ActiveForm.ActiveControl.Name`

В связи с тем, что объектная модель Visual FoxPro поддерживает наследование, при работе с классами не обойтись без соответствующей информационной поддержки. Для получения информации о созданных объектах можно воспользоваться следующими свойствами. Эти свойства имеют статус "только для чтения" и не могут использоваться для изменения свойств объекта.

Object.BaseClass

Возвращает имя базового класса, на основе которого создан указанный объект.

Object.Class

Возвращает имя класса.

Object.ClassLibrary

Возвращает имя файла пользовательской библиотеки классов, в которой содержится определение класса, на основе которого создан объект.

Object.ParentClass

Возвращает имя класса, который является родительским для класса, на основе которого создан указанный объект.

Control.Parent

Обеспечивает ссылку на содержащий элемент управления объект-контейнер. Например, мы можем изменить цвет фона формы при каких-либо действиях с включенным в нее элементом управления без необходимости знать имя этой формы:

`THIS.Parent.BackColor = RGB(192,0,0)`

Таким образом, главное назначение этого свойства - использование его при создании универсальных классов элементов управления, способных воздействовать на объект, в котором будут размещены элементы управления, созданные на основе этого класса.

Для почти любого объекта мы можем записать комментарий, что может оказаться очень полезным на этапе отладки или сопровождения программы. Необходимо воспользоваться свойством

`Object.Comment [= cExpression]`

Это свойство удобно использовать для дополнительной идентификации объектов, если нет

необходимости изменять установленные идентификаторы, доступные для СУБД.

Записать какие-либо данные для объекта можно с помощью еще одного свойства

Object.Tag [= *Expression*]

Оба перечисленных выше свойства по умолчанию, если им не присвоены какие-либо значения, возвращают пустую символьную строку.

Для работы с классами и созданными на их основе объектами из программы в Visual FoxPro существует несколько очень важных команд и функций, которые мы и рассмотрим. Мы советуем вам максимально использовать возможности визуальной работы с классами и объектами, которые описаны в [главе 10](#). В то же время мы сознательно уделяем этим командам и функциям столько внимания при описании объектной модели даже несмотря на то, что этот материал имеет специфическое отношение только к Visual FoxPro, так как если вы внимательно разберете приведенные здесь примеры, то в дальнейшем будете уверенно ориентироваться среди рассматриваемых в этом параграфе сложных понятий ООП.

ADD CLASS *ClassName* [OF *ClassLibraryName1*] TO *ClassLibraryName2* [OVERWRITE]

Добавляет описание класса в визуальную библиотеку классов. Параметр *Class-Name* определяет имя класса, добавляемого в визуальную библиотеку классов *ClassLibraryName2*. Если файл визуальной библиотеки классов не существует, Visual FoxPro создает визуальную библиотеку классов и добавляет в нее определение класса. Если вы опускаете необязательную опцию OF *ClassLibraryName1*, Visual FoxPro ищет описание класса в любых визуальных библиотеках классов, открытых командой **SET CLASSLIB**. Visual FoxPro сгенерирует ошибку, если определение класса не может быть размещено или определение класса с именем, которое вы задаете, уже существует в *ClassLibraryName2*. Опция OF *ClassLibraryName1* определяет визуальную библиотеку классов, из которой копируется определение класса. Опция **OVERWRITE** удаляет все классовые определения из визуальной библиотеки классов прежде, чем определение нового класса будет добавлено.

Использование команды **ADD CLASS** добавляет определение класса в библиотеку классов или копирует определение класса из одной визуальной библиотеки классов в другую. Определение класса не может быть добавлено из программы, процедурного файла или приложения Visual FoxPro (файлы с расширениями PRG или APP).

```
DEFINE CLASS ClassName1 AS ParentClass
  [[PROTECTED Property1, Property2 ...]
    Property = Expression...]
  [ADD OBJECT [PROTECTED] Object AS ClassName2 [NOINIT]
    [WITH cPropertyList]]...
  [[PROTECTED] FUNCTION | PROCEDURE Name
    [NODEFAULT]
    cStatements
  [ENDFUNC | ENDPROC]]...
ENDDDEFINE
```

Создает определяемый пользователем класс или подкласс и задает свойства, события и методы для класса или подкласса. Параметр *ClassName1* определяет имя создаваемого класса.

Опция **AS ParentClass** определяет родительский класс, на котором будет основан создаваемый класс или подкласс. Родительским классом может быть базовый класс Visual FoxPro, такой, например, как **Form** или любой другой определяемый пользователем класс или подкласс. Невизуальный определяемый пользователем класс может быть создан определением имени **Custom** для *ParentClass*.

С помощью опции [PROTECTED *Property1*, *Property2* ...] *Property* = *Expression*... можно назначить свойства создаваемому классу и установить для них значения, которые будут использоваться по умолчанию. Знак равенства говорит о том, что свойству *PropertyName* присваивается значение выражения *Expression*. Чтобы предотвратить доступ и изменение значений свойств вне определения класса или подкласса, включайте опцию **PROTECTED** и список защищенных свойств. Методы и события внутри определения класса или подкласса могут обращаться к защищенным свойствам.

Опция **ADD OBJECT** позволяет добавить объект к определению класса или подкласса из базового класса Visual FoxPro, определяемого пользователем класса или подкласса, либо из класса OLE. Параметр *Object* определяет имя объекта и используется для ссылки на объект внутри определения класса или подкласса после его создания. Параметр *ClassName* определяет имя класса или подкласса, содержащего объект, который вы добавляете к определению класса. Опция **NOINIT** указывает на то, что метод **Init** не выполняется при добавлении объекта.

Опция **WITH cPropertyList** определяет список свойств и значений свойств объекта, который вы добавляете к определению класса или подкласса.

Опции **FUNCTION *Name*** или **PROCEDURE *Name*** позволяют создать описание действий, выполняемых при возникновении события или выполнении метода для класса или подкласса. События и методы создаются как набор функций или процедур.

Включение опции **NODEFAULT** указывает на то, что **Visual FoxPro** не будет реагировать на события, как это должно было бы случиться, или выполнять процедуры обработки методов. Это позволяет использовать свои собственные обработчики определенных событий, отличающиеся от логики, принятой в **Visual FoxPro**. Опция **NODEFAULT** может располагаться в любом месте внутри процедуры обработки события или метода, но эта опция должна быть помещена внутри процедуры обработки события или метода в Конструкторе формы (**Form Designer**).

Параметр *cStatements* - это команды **Visual FoxPro**, которые выполняются, когда вызывается событие или метод.

Функции и процедуры событий и методов могут принимать значения путем включения оператора **PARAMETERS** как первой выполняемой строки в функцию или процедуру.

Чтобы создать объект на основе определения класса или подкласса, используйте функцию **CREATEOBJECT()** с именем соответствующего класса или подкласса.

Определения класса и подкласса, созданные с помощью команды **DEFINE CLASS**, не могут размещаться внутри команд структурного программирования типа **IF...ENDIF** или **DO CASE...ENDCASE** и в циклах типа **DO WHILE...ENDDO** или **FOR...ENDFOR**.

В качестве примера посмотрим, как используется эта команда для создания объектов. Создадим три различные формы.

```
oForm1 = CREATEOBJECT("frmTestForm")
oForm2 = CREATEOBJECT("frmTestForm")
oForm3 = CREATEOBJECT("frmTestForm")
* Изменим их заголовки
oForm1.Caption = "Первая форма"
oForm2.Caption = "Вторая форма"
oForm3.Caption = "Третья форма"
* Выведем на экран с помощью метода Show первую форму
oForm1.Show
* Немножко раздвинем их на экране
oForm2.Move(oForm1.Left + 50, oForm1.Top + 50)
oForm2.Show
oForm3.AutoCenter = .T.
oForm3.Show
* Подождем реакции пользователя
READ EVENTS
* Определим класс для создания наших форм
DEFINE CLASS frmTestForm AS Form
    BackColor = RGB(192,192,193)
    Caption = "TestForm"
    * Добавим в форму управляющую кнопку
    * для закрытия формы
    ADD OBJECT cmdExit AS CommandButton WITH ;
        Caption = "\<<Выход", ;
        Left = 150, ;
        Top = 100, ;
        AutoSize = .T.
    PROCEDURE cmdExit.Click
        RELEASE THISFORM
        * Когда с экрана будет убрана последняя форма,
        * отменим состояние ожидания
        IF _SCREEN.FormCount = 1
            CLEAR EVENTS
        ENDIF
    ENDPROC
ENDDEFINE
```

Следующая команда:

```
SET CLASSLIB TO ClassLibraryName [ADDITIVE]
[ALIAS AliasName]
```

Открывает визуальную библиотеку классов с определениями хранящихся в ней классов. Параметр *ClassLibraryName* определяет имя файла библиотеки. Опция **ADDITIVE** позволяет

открыть указанную библиотеку, не закрывая открытой ранее. Опция **ALIAS *AliasName*** позволяет задать псевдоним для библиотеки, на который можно ссылаться при создании объекта на базе класса, определение которого хранится в данной библиотеке. Задание этой команды в виде **SET CLASSLIB TO** закрывает все открытые библиотеки классов.

RELEASE CLASSLIB *ClassLibraryName*

Позволяет закрыть указанную визуальную библиотеку классов из открытых ранее.

Помимо команд при работе с классами в программе не обойтись без следующих функций.
CREATEOBJECT(*ClassName* [, *Parameter1*, *Parameter2*, ...])

Создает объект из описания класса или объекта OLE. Аргумент *ClassName* определяет класс или OLE-объект, из которого будет создан новый объект. Visual FoxPro ищет класс или OLE-объект в следующем порядке:

1. Базовые классы Visual FoxPro.
2. Определяемые пользователем описания классов в том порядке, в котором они были загружены в память.
3. Классы в текущей программе.
4. Классы в библиотеках классов, открытые с помощью **SET CLASSLIB**.
5. Классы в процедурах, открытых с помощью **SET PROCEDURE**.
6. Классы в последовательности выполнения программ Visual FoxPro.
7. Регистр Windows (для объектов OLE).

Для создания OLE-объектов используется следующий синтаксис параметра *ClassName*:
ApplicationName.Class

Например, для работы с таблицами Microsoft Excel с помощью средств OLE вы можете написать:

```
oExcelSheet = CREATEOBJECT("Excel.Application")
```

Когда этот код будет выполнен, запускается Microsoft Excel в скрытом для пользователя виде. Вы не сможете обнаружить его отображение на панели задач Windows 95. Но в перечне загруженных задач, появляющемся при нажатии клавиш **Ctrl+Alt+Del**, в этом случае пакет Excel присутствует. Заметьте также, что даже если пакет Excel загружен на компьютере, в скрытом виде загружается еще одна его копия. Подробнее работу OLE Automation мы обсудим в [десятой главе](#).

Необязательные параметры *Parameter1*, *Parameter2*, ... используются, чтобы передать значения в процедуру события **Init** для класса. Событие **Init** выполняется, когда вы используете функцию **CREATEOBJECT()** и она разрешает инициализацию объекта.

Используйте функцию **CREATEOBJECT()** для создания объекта из описания класса или объекта OLE и назначения ссылки на объект с помощью переменной или элемента массива. Прежде чем вы сможете создать объект из определяемого пользователем класса, необходимо его определить с помощью команды **DEFINE CLASS** или получить к нему доступ в визуальной библиотеке классов, открытой с помощью **SET CLASSLIB**. Используйте знак равенства или команду **STORE** для назначения ссылки на объект с помощью переменной или элемента массива.

В качестве примера с помощью этой функции давайте программно создадим несколько объектов на базе одного класса. В визуальной библиотеке классов у нас хранится описание класса панели инструментов. Создадим на его основе три панели и зададим им разные свойства.

```
* Открываем визуальную библиотеку классов
SET CLASSLIB TO Office
* Создаем три объекта
oTbr1=CREATEOBJECT("Office_Toolbar")
oTbr2=CREATEOBJECT("Office_Toolbar")
oTbr3=CREATEOBJECT("Office_Toolbar")
* Изменяем для каждой созданной панели заголовок ее окна
oTbr1.Caption = "Первая панель"
oTbr2.Caption = "Вторая панель"
oTbr3.Caption = "Третья панель"
* Для первой панели изменим цвет фона
oTbr1.BackColor = RGB(0,0,255)
* Выведем их на экран
oTbr1.Show
oTbr2.Show
oTbr3.Show
* Дадим 20 с для того, чтобы можно было их рассмотреть и
* подвигать
```

READ TIMEOUT 20

Для определения ссылки на создаваемые объекты удобно использовать массив, как это видно из следующего примера, в котором создается пять форм.

```
* Определяем массив для создания ссылки
PUBLIC ARRAY aTest(1)
DIMENSION aTest(5)
* Очищаем экран
_SCREEN.CLS
* Создаем пять форм и помещаем ссылки на них в массив
FOR nCount = 1 TO 5
  aTest(nCount) = CREATEOBJECT("frmTestForm")
  * Размещаем их по центру экрана
  aTest(nCount).AutoCenter = .T.
ENDFOR
* Выводим созданные формы на экран
FOR nCount = 1 TO 5
  IF TYPE("aTest(nCount)") = "O"
    aTest(nCount).Show
  ENDIF
ENDFOR
* Определяем класс для создаваемых форм с кнопкой
* для ее стирания
DEFINE CLASS frmTestForm AS Form
ADD OBJECT cmdExit As CommandButton WITH ;
  Caption = "<<Выход", ;
  Top = 111, ;
  Left = 108, ;
  Height = 29, ;
  Width = 94, ;
  Visible = .T.
PROCEDURE cmdExit.Click
  RELEASE ThisForm
ENDPROC
ENDDEFINE
```

Функция

AINSTANCE(ArrayName, cClassName)

Размещает все образцы класса в массиве. Аргумент *ArrayName* определяет имя массива, в котором размещаются образцы. Если массива, который вы определяете, не существует, Visual FoxPro автоматически создаст его. Если массив существует, но не достаточно большой, чтобы вместить все образцы, Visual FoxPro автоматически увеличит его размер. Если массив оказывается больше, чем необходимо, Visual FoxPro усечет его до нужных размеров. В случае, когда массив существует и функция **AINSTANCE()** возвращает 0 - никакие образцы не найдены, - массив остается неизменным. Аргумент *cClassName* определяет имя базового класса Visual FoxPro, пользовательского класса или объекта Visual FoxPro (Cursor, DataEnvironment, Relation). Функция **AINSTANCE()** возвращает количество образцов класса, размещенных в массиве.

AMEMBERS(ArrayName, Object [, 1 | 2])

Размещает в массиве для указанного объекта имена свойств, процедур и включенных объектов. Аргумент *ArrayName* определяет массив, в который записываются имена элементов свойств для *ObjectName*. Если массив не достаточно большой, чтобы вместить все имена, Visual FoxPro автоматически увеличивает его размер. Если вы определяете существующий двумерный массив, Visual FoxPro преобразует его в одномерный. Аргумент *Object* определяет объект, элементы свойств которого размещаются в массиве, указанном в *ArrayName*. Аргумент *Object* может представлять собой любое выражение, имеющее отношение к объекту, типа ссылки на объект, переменной объекта или элемента массива объекта. Аргумент 1 указывает на то, что в массив будут включены как свойства объекта, так и методы и включенные объекты. Массив является двумерным со вторым столбцом, уточняющим, к какому типу относится элемент, внесенный в список в первом столбце. Возможные значения для второго столбца: **Property**, **Event**, **Method** или **Object**. Аргумент 2 указывает на то, что массив будет содержать имена объектов, являющихся элементами объекта, указанного в *Object*. Массив является одномерным. Эта опция обеспечивает метод для определения имен всех форм в наборе форм или элементов управления в форме. Функция **AMEMBERS()** возвращает количество объектов, свойств и процедур для объекта, либо 0, если массив не может быть создан.

COMPOBJ(*oExpression1*, *oExpression2*)

Сравнивает свойства двух объектов и возвращает значение "истина" (.T.), если их свойства и значения свойств идентичны. Аргументы *oExpression1*, *oExpression2* определяют объекты для сравнения. Параметры *oExpression1* и *oExpression2* могут быть любыми определяющими объекты выражениями, такими как ссылки на объект, переменные объекта или элементы массива объекта.

Функция **COMPOBJ()** возвращает значение "ложь" (.F.), если объект имеет такие свойства, которых нет у другого объекта, или если объекты имеют идентичные свойства, но значения одного или нескольких свойств оказываются различными.

5.2. Объекты и их свойства

Если из материалов предыдущего параграфа вы сделали вывод, что основной смысл ООП заключается в создании объектов и наличии удобных средств манипуляции ими, то мы с вами почти безоговорочно соглашаемся. А поэтому и перейдем к более подробному обсуждению этих объектов.

В этом параграфе мы познакомимся с

- основными объектами и элементами управления для создания интерфейса пользователя;
- назначением отдельных объектов;
- основными свойствами объектов;
- компонентами для расширения возможностей разработчика - объектами **ActiveX**.

В **Visual FoxPro** создание объектов поддерживается стройной и мощной системой классов. На основе базовых классов программист может создать свои классы и, komponуя их в соответствующие библиотеки, обеспечивать их доступность для создания объектов и элементов управления в разрабатываемом приложении, как это показано на схеме, приведенной на рис. 5.3. Конечно, если быть совсем точным, эта схема существенно упрощена по сравнению с реальными возможностями. В распоряжении программиста, разрабатывающего приложение на **Visual FoxPro**, еще есть объекты для работы с данными при построении форм и отчетов, которые не имеют классов, и есть возможность использовать объекты других приложений и объекты **ActiveX**. Об этом мы поговорим в следующих главах.

В отличие от **Visual FoxPro**, в **Visual Basic** нет системы классов, и программист работает непосредственно с набором предоставляемых ему объектов. По аналогии с **Visual FoxPro** можно сказать, что на схеме, приведенной на рис. 5.3, в его распоряжении находятся первый и последний блоки.



Рис. 5.3.

В табл. 5.1 был приведен список объектов, которые программист может создать в **Visual FoxPro** на основе базовых классов. Визуально представляемые объекты легко создаются при проектировании форм простым перетаскиванием нужного класса объекта с панели инструментов **Form Control** на поверхность формы. Соответствующие кнопки этой панели инструментов можно видеть на рис. 5.4.

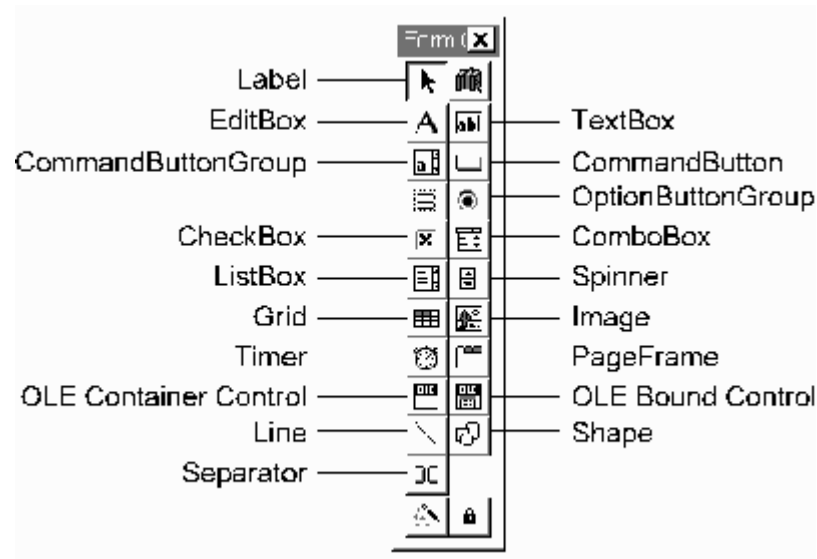


Рис. 5.4.

В Visual Basic аналогичные функции выполняет панель инструментов, которая называется **ToolBox**. Она содержит довольно похожий набор объектов и элементов управления (рис. 5.5). Причем, если в Visual FoxPro мы можем расширить набор объектов, используемых при разработке приложения за счет подключения дополнительных библиотек классов, то в Visual Basic дополнительные объекты просто включаются в панель инструментов **ToolBox**.

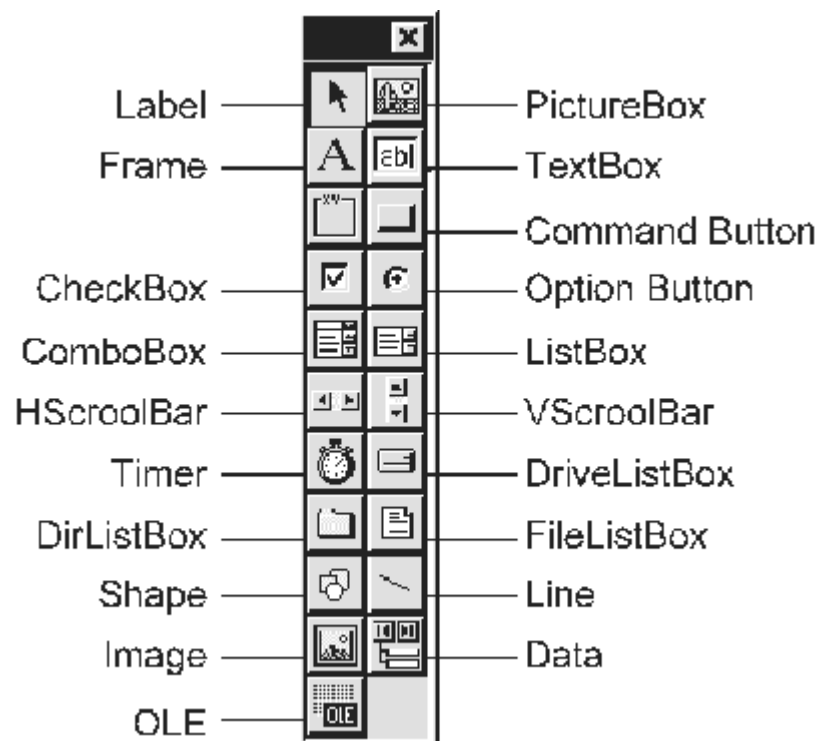


Рис. 5.5. Визуальные объекты и элементы управления в Visual Basic

Большинство объектов имеют достаточно общий набор свойств для определения их местоположения на экране или в форме, внешнего оформления и т. д. Отличия касаются в основном способов обеспечения тех особых функций, для которых собственно объект и существует в приложении.

Дадим краткий комментарий тех свойств, которые есть у большинства объектов.

Как вы уже заметили, чтобы что-то сделать с объектом, надо к нему обратиться. Каждый объект имеет уникальный идентификатор, который называется именем объекта. При создании объекта по умолчанию ему назначается имя, которое включает имя класса, на основе которого создается объект, и цифру по порядку включения в объект-контейнер. Например, при создании кнопки управления в форме она получит имя **CommandButton1**. Если мы создадим в этой форме еще одну кнопку, она получит имя **CommandButton2** и т. д. Чтобы изменить имя объекта, надо

использовать свойство

Object.Name [= *cName*]

Параметр определяет имя указанного объекта.

В [четвертой главе](#) мы уже останавливались на общепринятых соглашениях о наименовании переменных. В табл. 5.3 приведены рекомендации по наименованию объектов, которые основаны на стандартах Visual Basic. В соответствии с ними тип объекта (префикс) характеризуется тремя прописными буквами. Каждая смысловая часть в имени объекта начинается с большой буквы, но не отделяется знаком подчеркивания.

Таблица 5.3. Правила наименования объектов

Объект	Префикс	Пример
CheckBox	chk	chkReadOnly
Column	grc	grcCurrentPrice
ComboBox	cbo	cboEnglish
CommandButton	cmd	cmdCancel
CommandGroup	cmg	cmgChoices
Container	cnt	cntMoverList
Control	ctl	ctlFileList
Data Control	dat	datAvto
Drive List Box	drv	drvTarget
EditBox	edt	edtTextArea
File List Box	fil	filSource
Form	frm	frmFileOpen
FormSet	frs	frsDataEntry
Frame	fra	fraControls
Grid	grd	grdPrices
Header	grh	grhTotalInventory
HScrollBar	hsb	hsbVolume
Image	img	imgIcon
Label	lbl	lblHelpMessage
Line	lin	linVertical
ListBox	lst	lstPolicyCodes
OLE	ole	oleObject1
OLEBoundControl	olb	olbObject1
OptionButton	opt	optFrench
OptionGroup	opg	opgType
Page	pag	pagDataUpdate
PageFrame	pgf	pgfLeft
Separator	sep	sepToolSection1
Shape	shp	shpCircle
Spinner	spn	spnValues
TextBox	txt	txtGetText
Timer	tmr	tmrAlarm
ToolBar	tbr	tbrEditReport
VScrollBar	vsb	vsbVolume

Большинство объектов может отображать какой-либо текст. Мы имеем возможность выбрать вид шрифта для вывода текста с помощью свойства

Object.FontName [= *cName*]

Параметр *cName* определяет имя шрифта, который будет использоваться объектом с именем *Object*. По умолчанию будет использоваться шрифт Arial. Если вы изменяете шрифт и какие-либо его параметры, первоначально установите имя шрифта, а затем такие его параметры, как размер и начертание.

Object.FontSize [= *nSize*]

Устанавливает размер используемого шрифта в пунктах. По умолчанию используется шрифт размером 10 пунктов. Максимальное значение для *nSize* составляет 2048 пунктов. Напомним, что

1 мм составляет примерно 2,8 пункта.

Object.FontBold [= *IExpression*]

Устанавливает полужирное начертание шрифта, если параметр равен .Т.. Это значение принято по умолчанию.

Object.FontItalic [= *IExpression*]

Устанавливает наклонное начертание шрифта, если параметр равен .Т.. По умолчанию принято значение .F..

Object.FontStrikeThru [= *IExpression*]

Устанавливает перечеркнутое начертание шрифта, если параметр равен .Т.. По умолчанию принято значение .F..

Object.FontUnderline [= *IExpression*]

Устанавливает подчеркнутое начертание шрифта, если параметр равен .Т.. По умолчанию принято значение .F..

Практически все объекты могут иметь различный цвет. Для изменения цвета объекта, как при его создании, так и в процессе работы программы, можно использовать следующие свойства.

Object.BackColor [= *nColor*]

Задаёт цвет фона объекта. Параметр *nColor* представляет собой число, которое обозначает цвет. Вместо этого числа удобнее использовать функцию

RGB(*nRedValue*, *nGreenValue*, *nBlueValue*)

которая возвращает нужное значение *nColor* в зависимости от сочетания интенсивностей красного *nRedValue*, зеленого *nGreenValue* и синего *nBlueValue* цветов. Каждое значение интенсивности может меняться от 0 до 255. Максимальное значение обозначает наибольшую интенсивность цвета. В табл. 5.4 приведены значения для типичных цветов.

Таблица 5.4. Значения для типичных цветов

Цвета	Значения RGB	Значение nColor
Белый	255, 255, 255	16777215
Черный	0, 0, 0	0
Серый	192, 192, 192	12632256
Темно-серый	128, 128, 128	8421504
Красный	255, 0, 0	255
Темно-красный	128, 0, 0	128
Коричневый	255, 255, 0	65535
Бежевый	128, 128, 0	32896
Зеленый	0, 255, 0	65280
Темно-зеленый	0, 128, 0	32768
Морской волны	0, 255, 255	16776960
Темный морской волны	0, 128, 128	8421376
Синий	0, 0, 255	16711680
Темно-синий	0, 0, 128	8388608
Малиновый	255, 0, 255	16711935
Темно-малиновый	128, 0, 128	8388736

Object.ForeColor [= *nColor*]

Задаёт цвет выводимых символов.

Object.BorderColor [= *nColor*]

Задаёт цвет рамки.

Control.BorderWidth [= *nWidth*]

Для указанного элемента управления задаёт ширину рамки. Параметр *nWidth* может изменяться от 0 до 8192. Естественно, если параметр *nWidth* будет равен 0, рамка рисоваться не будет.

Кроме текста многие элементы управления могут выводить изображение. Например, достаточно часто какое-то изображение (пиктограмма) на кнопке управления воспринимается оператором легче, чем надпись. К тому же такой подход позволяет проще реализовать многоязычные приложения, так как пиктограмма в отличие от текста не требует перевода. В то же время следует учитывать, что использование изображений требует больших ресурсов компьютера. Для вывода изображений на элементах управления служит свойство

Control.Picture [= *cFileName* | *GeneralFieldName*]

Изображение может храниться либо в файле *cFileName*, либо в поле типа *General* *GeneralFieldName*. Изображение должно храниться в формате BMP или ICO.

Для объектов, которые не имеют обозначенных какими-либо видимыми элементами границ,

можно установить свой фон или сделать их прозрачными, то есть принять для них фон объекта-контейнера, на котором они располагаются, с помощью следующего свойства

Object.BackStyle [= *nStyle*]

Для того чтобы фон стал прозрачным, параметр *nStyle* должен иметь значение 0, при этом установка свойства BackColor игнорируется. Для установки собственного фона объекта параметр *nStyle* должен быть равен 1.

Наиболее часто используемые объекты для их идентификации пользователем имеют заголовки - надписи, размещаемые на элементе управления (кнопка управления, поле проверки) или вверху объекта (форма, страница). При создании такого нового объекта или элемента управления они получают по умолчанию заголовок такой же, как имя объекта. Изменить заголовок можно с помощью свойства

Object.Caption [= *cText*]

Параметр *cText* определяет текст заголовка для указанного объекта.

Для задания размеров объекта и его расположения можно использовать следующие свойства:

Object.Height [= *nValue*]

Параметр *nValue* определяет высоту указанного объекта.

[Object].Width [= *nValue*]

Параметр *nValue* определяет ширину указанного объекта.

Object.Left [= *nValue*]

Параметр *nValue* определяет расстояние от левого края объекта-контейнера до левой границы указанного объекта.

Object.Top [= *nValue*]

Параметр *nValue* определяет расстояние от верхнего края объекта-контейнера до верхней границы указанного объекта.

Единица измерения для параметра *nValue* в четырех вышеперечисленных свойствах определяется свойством

Object.ScaleMode = *nMode*

которое воздействует на форму или панель инструментов, а следовательно, действует и для всех расположенных в них объектах или элементах управления.

В Visual FoxPro если параметр *nMode* равен 0, то в качестве единицы измерения используется фоксель, если 3 - пиксель (минимальная точка, которая может быть отображена на экране). Фоксель - это единица измерения, которая используется в Visual FoxPro для облегчения разработки приложений, которые должны работать и в текстовой и в графической среде. Фоксель примерно соответствует максимальной высоте и средней ширине символа в используемом шрифте. Мы настоятельно рекомендуем вам использовать пиксели, так как в противном случае вы рискуете наблюдать свои формы в виде, несколько отличном от предполагаемого при разработке. Недостатком использования пикселей является то, что на разных системах дисплеев и при разном разрешении число пикселей, которые размещаются на экране, различно. Это ведет к тому, что созданный при одном разрешении экранный интерфейс будет занимать слишком мало места при более высоком разрешении или не уместиться на экране при более низком. Для приложений производственного использования приемлемым решением является ограничение пользователя в выборе разрешения экрана. Если это не приемлемо, придется прибегнуть к специальным методам проектирования пользовательского интерфейса, обеспечивающим отслеживание установленного разрешения монитора и автоматическую коррекцию выводимых программой экранных элементов.

В Visual Basic параметр *nMode* может принимать следующие значения:

- 0 - используется собственная координатная система;
- 1 - twip - единица измерения, которая обеспечивает сохранение пропорций пользовательского интерфейса независимо от используемых систем дисплеев;
- 2 - типографский пункт (то же, что для шрифтов);
- 3 - пиксель;
- 4 - символ (120 twip по горизонтали и 240 - по вертикали);
- 5 - дюйм;
- 6 - миллиметр;
- 7 - сантиметр.

Пользователю значительно помогает ориентироваться в интерфейсе прикладной программы возможность получения оперативной подсказки о назначении того или иного элемента управления. Текст такой подсказки, которая появляется, если пользователь задержит указатель мыши на элементе управления, определяется свойством

Control.ToolTipText = *cText*

Параметр должен содержать подсказку для указанного элемента управления.

Для создания ссылки в файле контекстной помощи в пользовательской программе можно использовать следующее свойство

Object.HelpContextID [= *nContextID*]

Параметр *nContextID* определяет идентификатор темы, содержание которой выводится на экран, если пользователь нажимает клавишу **F1**, когда активен указанный объект.

Естественно, мы используем в программе элементы управления, чтобы узнать желание пользователя выполнить то или иное действие. Определить этот выбор помогает свойство **Control.Value [= *nSetting*]**

которое возвращает информацию о состоянии указанного элемента управления. При более подробном описании каждого объекта мы еще остановимся на особенностях использования этого свойства. Если мы предварительно хотим установить для элемента управления какое-то значение с помощью параметра *nSetting*, то следует соблюдать правильный тип данных, соответствующий элементу управления.

Object.Visible [= *IExpression*]

Позволяет сделать указанный объект невидимым на экране, если выражение *IExpression* будет иметь значение **.F.** При создании объекта в Конструкторе формы свойство по умолчанию имеет значение **.T.**, в программе - **.F.** При добавлении объекта в форму, пока происходит определение его свойств или при изменении сразу нескольких характеристик объекта, чтобы избежать многочисленных перерисовок, установите свойство **Visible** равным **.F.**

Object.Enabled [= *IExpression*]

Позволяет сделать объект недоступным для изменения пользователем, если выражение *IExpression* будет иметь значение **.F.** При этом данный объект перестает реагировать на какие-либо события и может использоваться, как например, **TextBox** или **EditBox**, только для отображения выводимых в него данных.

Управлять порядком перемещения пользователя по элементам управления можно с помощью свойства

Control.TabIndex [= *nOrder*]

Параметр определяет номер элемента управления в объекте-контейнере (например, в форме), в соответствии с которым по умолчанию будет регулироваться порядок перехода пользователя к следующему элементу управления.

Если вы заранее знаете, что будущие пользователи не любят использовать мышь, то установить действие, происходящее для элемента управления при нажатии клавиши **Tab** можно с помощью свойства

Control.TabStop [= *IExpression*]

Выражение *IExpression* определяет, включен ли элемент управления **Control** в стек свойства **TabIndex** для перехода между элементами управления с помощью клавиши **Tab**. Если выражение *IExpression* равно **.F.**, то элемент управления в стек не включен, и при нажатии клавиши **Tab** этот элемент пропускается.

Конечно, при создании приложения для обработки данных нас больше всего волнуют объекты, прямо для этого предназначенные. Поэтому более подробный разговор начнем именно с таких объектов.

Объекты для работы с данными

Наиболее универсальным объектом для работы с данными, бесспорно, является **Text Box** - *текстовое поле*. С его помощью можно отображать и редактировать данные любого типа, кроме, пожалуй, изображений.

Для того чтобы элемент управления отображал какие-то данные или изменял свое значение в зависимости от каких-либо условий, его надо привязать к переменной или полю в таблице.

Сделать это в **Visual FoxPro** можно с помощью свойства

Object.ControlSource [= *cName*]

В параметре *cName* указывается имя переменной или поля в таблице.

В **Visual Basic** для того, чтобы привязать какой-либо объект к данным в БД, необходимо сначала создать объект **Data** - **элемент управления для работы с БД** и указать для него в свойстве

Object.DatabaseName [= *cName*]

имя и путь к базе данных. Если в качестве источника данных используется формат, в котором таблицы хранятся в отдельных файлах и не объединены на постоянной основе, например **FoxPro 2.x**, то указывается только путь к папке с этими файлами.

В число обязательных действий включается установка значений еще для двух свойств. Сначала выберите значение для свойства

Object.RecordsetType [= *Value*]

Процессор баз данных **Microsoft Jet** допускает выбор из трех вариантов:

0 - **tables** - позволяет работать непосредственно с таблицами. Этот вариант допустим только для **Access** или БД, поддерживаемых драйверами **ISAM**. Но его нельзя использовать, например, для работы с БД, расположенной на **MS SQL Server**.

1 - **dynasets** - позволяет создать обновляемый набор записей, выбирая и считывая из источника данных только необходимые записи на основе уникальных ключевых значений для каждой записи. Этот способ позволяет снизить объем данных, передаваемых на рабочую станцию с сервера. Данные из полей примечаний и изображения считываются только тогда, когда возникает необходимость их отображения на экране.

2 - **snapshots** - позволяет создать набор записей, которые не будут иметь возможности обновления в источнике данных.

После этого вы можете выбрать нужную для работы таблицу с помощью свойства **Object.RecordSource** [= *Value*]

В качестве параметра *Value* для этого свойства можно использовать имя таблицы, символьное выражение, содержащее команду SQL, приемлемую для используемой БД, или имя одного из объектов **QueryDef**, содержащегося в коллекции **QueryDefs** объекта **Database** (об объектах для доступа к данным, используемых в процессоре баз данных Microsoft Jet, см. главу 6).

Вот теперь, вернувшись к объекту **TextBox** и предварительно указав в свойстве **DataSource** имя элемента управления для работы с БД, мы можем назначить для него требуемый источник данных с помощью свойства

Object.DataField [= *cName*]

В качестве параметра *cName* вы можете указать имя поля, данные из которого хотите отобразить в текстовом поле.

Для вывода и редактирования больших объемов текстовой информации, которая может храниться, например, в полях примечаний, в **Visual FoxPro** есть специальный элемент управления - **EditBox** - поле редактирования. В поле редактирования доступны все возможности по работе с текстом, такие как вырезка, копирование и т. д. Текст в поле редактирования может прокручиваться по вертикали, а длинные строчки автоматически усекаются по правой границе поля и переносятся на следующую строку.

В **Visual Basic** для этого используется текстовое поле, которое имеет специальное свойство **Multiline**. При установке его значения в **True** текстовое поле может занимать несколько строк.

В том случае, если пользователь должен выбрать какие-то значения из списка, лучше всего использовать элементы управления **ComboBox** - раскрывающийся список или **ListBox** - список.

В случае, если необходимо так организовать ввод данных, чтобы дать пользователю возможность выбирать данные только из заранее определенного списка, следует присвоить свойству **Style** значение 2 (drop-down list). Комбинированный список с возможностью ввода создается при значении свойства **Style** равным 0 (drop-down combo). В **Visual Basic** вы можете использовать еще одно значение - 1. При этом создается список **Simple Combo**, состоящий из текстового поля, в которое пользователь может вводить данные, и незакрывающегося списка.

В комбинированный список можно включать самые разнообразные данные. В **Visual Basic** источник данных будет определяться по такой же схеме, как для текстового поля. В **Visual FoxPro** тип определяется значением свойства **RowSourceType**.

Control.RowSourceType [= *nSource*]

Значения параметра *nSource* могут быть:

- 0 - (по умолчанию) - нет данных. Список заполняется во время работы пользовательской программы с помощью методов **AddItem** или **AddListItem**.
- 1 - значения. Список заполняется данными, непосредственно указанными в свойстве **RowSource**.
- 2 - псевдоним. Список заполняется данными из полей в таблице, открытой в указанной рабочей области.
- 3 - операторы SQL.
- 4 - запрос. В список помещаются результаты выполнения файла запроса (QPR).
- 5 - массив.
- 6 - поля. В отличие от значения 2, поля в список можно включить в произвольном порядке и из различных таблиц.
- 7 - файлы.
- 8 - структура таблицы.

Для заполнения списка используются значения, задаваемые свойством **Control.RowSource** [= *cList*]

Параметр *cList* может представлять, в зависимости от значения свойства **RowSourceType**, разделенный запятыми список значений, таблиц, файлов, операторы SQL, имя массива или имя файла запроса. Перечень файлов можно задать, используя символы шаблона.

В **Visual Basic** для вывода списков файлов, доступных устройств или папок есть специальные объекты - **FileListBox**, **DirListBox** и **DriveListBox**. Комбинация этих элементов дает возможность интерактивного выбора пользователем каких-либо файлов или места записи данных во время работы пользовательского приложения.

В первых двух список выводимых файлов или папок определяется свойством **Path**.

Данные, которые принимают одно из двух значений, наиболее удобно отображать с помощью элемента управления *CheckBox* - поле проверки.

Свойство *Value* этого элемента управления может принимать значение "истина" .Т. (1) или "ложь" .F. (0). Например, при изменении данных можно создать копию файла с данными до изменения, а можно не создавать. Программно для поля проверки можно задать и третье, неопределенное состояние, когда элемент управления не находится ни в первом, ни во втором состоянии. При этом свойство *Value* будет иметь значение *NULL* или 2.

Все вышеупомянутые объекты способны отображать данные из одного поля. Исключением являются списки, так как они могут отображать несколько колонок, а следовательно, и полей с данными. Но все-таки исключительно любимым программистами и, что особенно важно, пользователями способом представления данных остается таблица. Для создания таблицы проще всего использовать специально предназначенный для этого объект - *Grid*.

В отличие от *Visual FoxPro* в *Visual Basic* нет "родного" объекта *Grid*. Для создания таблицы вы можете использовать поставляемые вместе с *Visual Basic* дополнительные элементы управления - *ActiveX*. Об этих объектах мы расскажем в отдельном разделе этого параграфа.

Объект *Grid* является объектом-контейнером, который содержит объект-контейнер *Columns*, содержащий, в свою очередь, объект *Header*. Объекты *Columns* и *Header* тоже имеют свои свойства, события и методы. По умолчанию данные в каждой колонке отображаются с помощью текстового поля, но мы можем использовать для этого любой другой элемент управления, способный работать с данными. Например, ничто не мешает нам в каждой клетке *Grid* разместить еще один *Grid*.

Объект *Grid* является элементом управления, который позволяет эффективно работать одновременно с несколькими строками данных и является функциональным эквивалентом *Browse*. В *Grid* мы можем помещать данные как из таблиц, так и из просмотров и курсоров или запросов. Целый ряд свойств позволяет динамически управлять *Grid* в зависимости от самых разнообразных условий. Например, мы можем легко обеспечить такую экзотическую функциональность, как вывод данных различным цветом в зависимости от их значения.

Для работы с данными в *Grid* можно использовать следующие свойства.

Grid.RecordSourceType [= *nType*]

Определяет тип источника данных для заполнения *Grid*. Параметр *nType* может принимать следующие значения:

- 0 - таблица. Автоматически открывается таблица, указанная в свойстве *RecordSource*.
- 1 - псевдоним (по умолчанию).
- 2 - по выбору пользователя. Источник данных устанавливает пользователь во время работы программы.
- 3 - запрос. В свойстве *RecordSource* должно быть указано имя файла-запроса (*QPR*).

Grid.RecordSource [= *cName*]

Определяет имя источника данных. Чаще всего это псевдоним курсора или таблицы.

Специальный набор свойств позволяет изменять параметры, установленные для колонки во время работы программы при каждом обновлении *Grid*. К таким свойствам относятся:

- *DynamicAlignment* - выравнивание данных;
- *DynamicCurrentControl* - используемый элемент управления;
- *DynamicForeColor* - цвет символов;
- *DynamicBackColor* - цвет фона;
- *DynamicFontName* - имя шрифта;
- *DynamicFontSize* - размер шрифта;
- *DynamicFontBold* - полужирное начертание шрифта;
- *DynamicFontItalic* - курсивное начертание шрифта;
- *DynamicFontStrikeThru* - перечеркнутое начертание шрифта;
- *DynamicFontUnderline* - подчеркнутое начертание шрифта.

Объекты для управления работой приложения

В эту группу объектов выделим средства создания различного типа кнопок, с помощью которых пользователь может выполнить какое-либо действие или вариант работы приложения. Здесь сразу придется оговориться, что выбор подобных действий может быть организован с помощью практически любого элемента управления, включая и такие, как текстовое поле, изображения, линии и т. д. - лишь бы такой элемент управления реагировал на какие-нибудь события. Эту группу объектов мы выделили исключительно из-за того, что управление работой приложения является их главным предназначением и их трудно использовать, например, для

работы с данными.

Наиболее часто используемым объектом для управления работой приложения, несомненно, является элемент управления **CommandButton** - *кнопка управления*.

Объект **CommandButton** создает отдельную управляющую кнопку. Управляющая кнопка обычно используется для запуска процедуры или события, которые реализуют какие-либо действия типа закрытия формы, перемещения к другой записи в таблице и т. д.

На кнопку мы можем поместить поясняющий текст с помощью свойства **Caption** или изображение (только в **Visual FoxPro**) с помощью свойства **Picture**.

Чтобы не задумываться над размерами кнопки, особенно если вы интерактивно меняете на ней текст, в **Visual FoxPro** установите для свойства **AutoSize** значение **.T.**

Как правило, действие, которое должно выполниться при нажатии на управляющую кнопку, определяется событием **Click**. Мы можем предусмотреть выполнение этого события и при нажатии клавиши **Enter**, для одной из двух или более управляющих кнопок в форме, когда активен какой-либо другой элемент управления, с помощью свойства **CommandButton.Default [= IExpression]**

Если для указанной кнопки параметр **IExpression** равен **.T.**, то эта кнопка сработает при нажатии клавиши **Enter** на любом другом элементе управления в форме. По умолчанию значение **IExpression** равно **.F.**. Естественно, только одна кнопка в форме может иметь значение **IExpression**, равное **.T.**

Совершенно аналогично для одной из управляющих кнопок в форме мы можем предусмотреть выполнение присвоенного ей действия при нажатии клавиши **Esc** с помощью свойства **CommandButton.Cancel [= IExpression]**

Для выбора какого-то одного действия из нескольких возможных лучше всего использовать кнопки выбора - **OptionButton**.

Обратите внимание, что кнопку выбора мы можем использовать только как объект для создания группы кнопок выбора. Если вы используете визуальные средства проектирования, то кнопка выбора как объект может использоваться в **Visual FoxPro** только в Конструкторе класса.

При создании кнопки выбора мы можем расположить поясняющий текст справа, как принято по умолчанию, или слева от изображения кнопки, присвоив свойству **Alignment** значение **1**.

Стоит отметить, что в **Visual FoxPro** для создания сразу нескольких кнопок существуют соответствующие объекты **CommandButtonGroup** и **OptionButtonGroup**. Их применение существенно ускоряет процесс создания экранных форм с постоянным набором управляющих действий.

Объект **Timer** - *таймер (счетчик времени)* - позволяет задать интервал времени или продолжительность выполнения каких-либо действий в программе. В отличие от других элементов управления, этот объект не виден пользователю во время работы программы. Его назначение заключается только в обеспечении функциональности пользовательского приложения, связанной с продолжительностью работы.

Основным свойством для таймера, которое позволяет задать интервал времени между его срабатыванием (выполнением события **Timer**), является **Timer.Interval [= nTime]**

По умолчанию значение **nTime** равно **0**, что препятствует срабатыванию таймера. Максимальное значение параметра **nTime** может быть равным **2147483647** миллисекунд, что превышает продолжительность **24** дней. При выборе значения для интервала срабатывания таймера необходимо учитывать следующее:

- В связи с тем, что система вырабатывает прерывания с частотой **18** раз в секунду, реальное значение интервала не может быть меньше, чем **56** миллисекунд.
- Установленное значение интервала может не соблюдаться, если система выполняет какие-либо длительные действия, связанные с большой загрузкой процессора (чтение и запись данных, интенсивные вычисления, сетевой доступ и т. д.). В этом случае событие **Timer** может наступить только после их завершения.
- Чем меньшее значение вы устанавливаете для интервала, тем чаще программа должна генерировать событие **Timer** и, следовательно, меньше ресурсов будет оставаться на выполнение других операций, и скорость работы программы может существенно уменьшиться.

Для сброса времени отсчета интервала и установки его отсчета с нуля используется метод **Reset**.

В **Visual Basic** своеобразный элемент управления в виде движка может быть выполнен на основе объектов **HScrollBar** и **VScrollBar**. Эти же объекты могут быть использованы для прокрутки длинных списков или больших объемов данных.

Объекты для оформления интерфейса пользователя

Объект **Image** создает элемент управления в виде изображения формата BMP. Может использоваться для создания областей на форме, реагирующих на действия пользователя.

В Visual Basic кроме объекта **Image** есть и объект **PictureBox**, который обладает расширенной функциональностью, например, может выводить дополнительно изображения в формате ICO и WMF, но работает медленнее.

Источник изображения для этого объекта устанавливается с помощью свойства

Control.Picture = *cFileName*

В Visual FoxPro источник изображения может храниться и в поле типа **General**. В подобном случае имя этого поля указывается вместо имени файла.

Объект **Label** создает метку для вывода поясняющего текста. Для придания метке нужного вида можно использовать следующие полезные свойства.

Control.Alignment [= *nValue*]

Если параметр *nValue* будет равен 0 (по умолчанию), то текст в метке выравнивается по левому краю, если 1 - по правому, если 2 - по центру.

Label.WordWrap [= *lExpression*]

Позволяет регулировать процесс изменения текста, заполняющего метку при изменении ее размера. По умолчанию параметр *lExpression* равен .F. и текст в метке не переносится между строками, при этом горизонтальный размер метки изменяется так, чтобы вместить текст по длине, а вертикальный остается без изменения, чтобы вместить имеющееся число строк текста с учетом размера шрифта. Если параметр *lExpression* равен .T., то текст может переноситься между строками с изменением вертикального размера метки в зависимости от длины текста и размера шрифта. При этом свойство **AutoSize** игнорируется.

Object.BorderStyle [= *nStyle*]

Это свойство определяет вид рамки для метки. По умолчанию параметр *nStyle* равен 0 и рамка не изображается. Если параметр *nStyle* равен 1, то метку будет окаймлять одинарная линия.

Очень полезное свойство **AutoSize** позволяет не отслеживать размер помещаемого в элемент управления текста, особенно если текст изменяется во время работы программы.

Объект **Line** создает элемент управления в виде линии, а **Shape** - в виде прямоугольника, окружности или овала. Требуемая форма последнего объекта в Visual FoxPro определяется свойством

Shape.Curvature [= *nCurve*]

По умолчанию параметр *nCurve* равен 0 и рисуется прямоугольник. Если значение параметра *nCurve* находится в диапазоне от 1 до 98, то будет рисоваться прямоугольник со все более скругленными углами, если 99 - круг или эллипс.

В Visual Basic форма объекта определяется другим свойством и более жестко

Object.Shape [= *Value*]

Параметр может принимать одно из следующих значений:

- 0 - прямоугольник (по умолчанию);
- 1 - квадрат;
- 2 - овал;
- 3 - окружность;
- 4 - прямоугольник со скругленными углами;
- 5 - квадрат со скругленными углами.

В Visual Basic объекты **Line** и **Shape** выполняют чисто оформительские функции, так как не реагируют на события. Аналогом объекта **Shape**, как элемента управления, в Visual Basic является **Frame**, который графически представляется прямоугольником с заголовком на верхней стороне и имеет необходимый набор событий.

Объекты-контейнеры

Одним из наиболее важных объектов, который используется для объединения всех элементов управления, размещаемых в одном окне, является объект **Form** - экранная форма. В целом интерфейс приложения состоит из форм, которые и обеспечивают ему требуемую функциональность при решении какой-либо задачи, например, редактирования или ввода данных, поиска нужной информации и т. д. Наиболее эффективным средством создания формы является Конструктор формы (Form Designer), о работе с которым пойдет речь в [главе 9](#).

Внешний вид формы может быть установлен с помощью большинства свойств, общих для всех элементов управления. Следующий набор свойств позволяет установить или убрать возможность управления формой как окном с помощью системного меню.

Object.ControlBox [= *IExpression*]

Определяет, появляется ли системное меню в верхнем левом углу формы во время ее выполнения. Если параметр *IExpression* принимает значение .Т. (по умолчанию), то системное меню отображается, если .F., то нет.

Object.MaxButton [= *IExpression*]

Определяет, имеет ли форма кнопку Maximize (увеличение размера окна до максимального размера). Если параметр *IExpression* принимает значение .Т. (по умолчанию), то форма имеет кнопку Maximize; если .F., то нет. Максимальное открытие окна Form во время выполнения программы вызывает событие Resize.

Object.MinButton [= *IExpression*]

Определяет, имеет ли форма кнопку Minimize (уменьшение размера окна до пиктограммы). Если параметр *IExpression* равен .Т. (по умолчанию), то форма имеет кнопку Minimize; если .F., то нет. Уменьшение формы до пиктограммы во время выполнения программы генерирует событие Resize.

Object.Icon [= *cFileName*]

Определяет изображение, которое будет выведено при минимизации формы. Файл, имя которого указано в *cFileName*, должен иметь по умолчанию расширение ICO.

Object.KeyPreview [= *IExpression*]

Определяет, прерывает ли событие KeyPress в форме событие KeyPress в элементе управления. Если параметр *IExpression* принимает значение .Т., то сначала событие KeyPress получает форма, а затем активный элемент управления; если .F. (по умолчанию), то событие KeyPress получает активный элемент управления, а форма не получает.

Вы можете использовать это свойство, чтобы создать для формы процедуру управления с помощью клавиатуры. Например, когда прикладная программа использует функциональные клавиши, вы можете обрабатывать нажатия клавиши на уровне формы скорее, чем для каждого элемента управления.

Объект **FormSet** создает набор форм, что позволяет легко координировать работу сразу с несколькими формами. Этот объект есть только в Visual FoxPro, в Visual Basic ему в соответствие можно поставить коллекцию **Forms**. Набор форм обеспечивает следующие возможности:

- Одновременное выполнение какого-либо действия сразу для нескольких форм.
- Свободное управление расположением форм на экране для достижения максимального удобства в работе с ними.
- Создание одного объекта DataEnvironment для всех форм в наборе и обеспечение тем самым синхронизации в перемещении данных.

В то же время следует заметить, что наличие в Visual FoxPro такого объекта, как страничный блок (PageFrame), позволяет вместить практически неограниченный объем данных в одну форму.

Для набора форм можно использовать множество свойств, событий и методов, доступных для отдельной формы. Некоторые свойства для набора форм особенно важны. Так, в Visual FoxPro для набора форм существует свойство

Object.WindowType [= nType]

Определяет поведение форм в наборе форм, если она запущена командой **DO FORM**. Для FormSet доступны следующие значения параметра *Type*:

nType	Описание
0	Независимо от режима.
1	Модально. Никакие объекты других форм не могут стать активными, и меню не доступно. Все формы в FormSet активны.
2	Чтение. Объект FormSet ведет себя, как будто он был активизирован командой READ . Выполнение останавливается на методе Show или команде DO FORM . Когда форма деактивирована, выполнение программы продолжается. Включено для совместимости снизу вверх.
3	Модальное чтение. FormSet ведет себя, как будто он был активизирован командой READ MODAL . Выполнение программы останавливается на методе Show или команде DO FORM . Любые объекты формы в FormSet и объектах формы, указанных в свойстве WindowList, доступны, но объекты других форм и меню не доступны. Включено для совместимости снизу вверх.

В Visual FoxPro в набор стандартных объектов входят и объекты для создания многостраничных форм.

Объект **Page** создает страницу в страничном блоке для размещения блока данных. Страница объединяет информацию, одновременно видимую на экране пользователем. Перелистывая страницы, пользователь может работать с большим объемом данных без необходимости перехода в другие окна или формы.

Страница как объект может входить только в страничный блок (**PageFrame**). В свою очередь, страница является объектом-контейнером и для определения количества включенных в нее элементов управления можно использовать свойство **ControlCount**. Для ссылки на отдельный элемент управления по его номеру и задания для него определенных значений свойств используется свойство **Controls**.

Object.KeyPreview [= IExpression]

Определяет, будет ли событие KeyPress в элементе управления прервано событием KeyPress в форме. Если параметр *IExpression* принимает значение .T., то сначала событие KeyPress получает форма, а затем активный элемент управления; если .F. (по умолчанию), то событие KeyPress получает активный элемент управления, а форма не получает.

Вы можете использовать это свойство, чтобы создать для страницы в форме процедуру управления с помощью клавиатуры. Например, когда прикладная программа использует функциональные клавиши, вы можете обрабатывать нажатия клавиши на уровне формы скорее, чем для каждого элемента управления.

Page.PageOrder [= nOrder]

Определяет относительный номер страницы в страничном блоке, что будет влиять на порядок их расположения.

Объект **PageFrame** создает страничный блок из набора страниц. Страничный блок позволяет компактно расположить большие объемы данных для работы с ними пользователя за счет размещения их на отдельных страницах, которые могут перелистываться. Узнать номер активной в данный момент страницы можно с помощью свойства **ActivePage**.

Объект **ToolBar** создает в Visual FoxPro панель инструментов, которая может облегчить пользователю выполнение часто повторяющихся действий. На панели инструментов мы можем размещать любые элементы управления. Это позволяет создавать универсальный набор элементов управления для использования с однотипными формами, например справочниками,

что сокращает время разработки программы и облегчает обучение пользователей.

Панель инструментов имеет несколько специфических свойств.

Object.Movable [= *IExpression*]

Определяет, может ли панель инструментов перемещаться пользователем во время выполнения программы. Если параметр *IExpression* принимает значение .Т. (по умолчанию), то панель инструментов может перемещаться. Если .F., то объект не может быть перемещен пользователем.

Object.Sizable = *IExpression*

Определяет возможность изменения размеров панели инструментов. По умолчанию параметр *IExpression* равен .Т., и размеры панели инструментов могут быть изменены пользователем. Если параметр *IExpression* будет равен .F., то сделать этого будет нельзя.

ToolBar.Docked [= *IExpression*]

Возвращает логическое значение, с помощью которого можно определить, встроена панель инструментов в рамку окна Visual FoxPro (.Т.) или нет (.F.). Если используется параметр *IExpression*, то свойство возвращает результат сравнения возвращаемого значения и параметра *IExpression*.

Если вы хотите поэкспериментировать со свойствами панели инструментов, задавая команды из окна *Command*, то вам поможет следующий образец, который обеспечивает доступ к панели инструментов, добавленной к форме при ее проектировании.

```
?_SCREEN.ActiveForm.Parent.Office_toolbar1.Docked
```

При добавлении панели инструментов к форме создается набор форм, поэтому для ссылки на него, как на объект более высокого уровня, используется указатель *Parent*.

nPosition = *ToolBar.DockPosition*

Определяет положение панели инструментов в окне Visual FoxPro. Параметр *nPosition* может принимать следующие значения: -1 - панель инструментов не может быть встроена; 0 - встроена в верхнюю рамку; 1 - в левую; 2 - в правую; 3 - в нижнюю.

На панели инструментов можно размещать специальный объект - **Separator**, который помещает пробел между элементами управления. Это позволяет визуально создавать на панели инструментов группы элементов управления, так как по умолчанию размещаемые на панели инструментов элементы управления следуют вплотную друг за другом.

Универсальными объектами в Visual FoxPro для создания компонентных элементов управления, включающих несколько отдельных объектов, являются объекты **Control** и **Container**. Об отличиях между этими объектами мы уже рассказывали.

Невизуальные объекты

Объект **Custom** создает единственный в Visual FoxPro невизуальный объект, исключая уже упоминавшиеся объекты **Container** и **Control**, которые выполняют специфические функции объединения других элементов управления.

Этот объект удобно использовать для создания и последующего запуска стандартных, часто используемых процедур, при создании которых не требуется визуализация объекта.

В качестве примера использования объекта **Custom** рассмотрим следующий пример, в котором мы создадим пользовательский класс, зашифровывающий пароль. Пароль может храниться в таблице с данными о доступе пользователей к элементам системы в зашифрованном виде. После ввода пароля пользователем хранимый пароль расшифровывается и сравнивается с введенным. В приводимой программе создается форма, в которой в поле, расположенное в левом верхнем углу, вводится пароль. Вводимые символы скрыты, и их число отображается звездочками. Нажмите кнопку "Зашифровать", и вы увидите в нижних полях формы слева зашифрованное значение, а справа введенное. Успехов в шифровании!

*** Пример использования объекта **Custom** для шифрования пароля

```

frmPwdForm = CREATEOBJECT("PwdForm") && Создаем форму.
frmPwdForm.Visible = .T.
READ EVENTS
* Описываем класс формы
DEFINE CLASS PwdForm AS FORM
    Caption = "Шифрование пароля"
* Добавляем объект в форму для формирования пароля
    ADD OBJECT CustPassword AS Pass_Word
        Height = 130
        Width = 350
        Autocenter = .T.
        * Добавляем текстовое поле для ввода пароля
        ADD OBJECT txtText1 AS TextBox WITH ;
            Height = 25, ;
            Left = 25, ;
            Top = 20, ;
            Width = 125, ;
            PasswordChar = "*", ;
            Name = "txtText1"
        * Добавляем текстовое поле для вывода
        * зашифрованного пароля
        ADD OBJECT txtText2 AS TextBox WITH ;
            Height = 25, ;
            Left = 25, ;
            Top = 72, ;
            Width = 125, ;
            Readonly = .T., ;
            Name = "txtText2"
        * Добавляем в форму кнопку для вызова процедуры
        * шифрования,
        * которая содержится в cmdCommand1.Click
        ADD OBJECT cmdCommand1 AS CommandButton WITH ;
            Top = 20, ;
            Left = 200, ;
            Height = 29, ;
            Width = 125, ;
            Caption = "Зашифровать", ;
            Name = "cmdCommand1"
        * Добавляем в форму текстовое поле для вывода введенного
        * пароля
        ADD OBJECT txtText3 AS TextBox WITH ;
            ControlSource = ; "THISFORM.custPassword.cUnencrypted", ;
            Height = 25, ;
            Left = 200, ;
            Readonly = .T., ;
            Top = 72, ;
            Width = 125, ;
            Name = "Text3"
        * Процедура, выполняемая после набора пароля
        PROCEDURE txtText1.LostFocus
            THISFORM.custPassword.Encrypt_It(TRIM(THIS.Value))
            THISFORM.txtText2.Value =;
            THISFORM.custPassword.cEncrypted
            THISFORM.cmdCommand1.SetFocus
        ENDPROC
        * Процедура, выполняемая при нажатии на кнопку
        * "Зашифровать"
        PROCEDURE cmdCommand1.Click
            THISFORM.custPassword.Decrypt_It(THISFORM.custPassword.; cEncrypted)
            THISFORM.Refresh
            THISFORM.txtText1.SetFocus
        ENDPROC
        * Процедура, выполняемая при выходе из формы
        PROCEDURE DESTROY
            CLEAR EVENTS
        ENDPROC

```

```

ENDDEFINE
* Описываем класс объекта Custom
DEFINE CLASS Pass_Word AS Custom
    Height = 17
    Width = 100
    Name = "Password"
    * Добавляем для создаваемого объекта свои свойства
    cUnencrypted = ""
    cEncrypted = ""
* Процедура выполнения шифрования пароля
    PROCEDURE Encrypt_It
        PARAMETERS cPassword
        cEncrypted_password = " "
        * Запускаем цикл по количеству введенных в пароле
        * символов
        FOR i = 1 TO LEN(cPassword)
            cLetter = SUBSTR(cPassword, i, 1)
            cEncrypted_password = cEncrypted_password + ;
            CHR((ASC(cLetter)*2)+5) && Заменяем
            && введенный символ
        NEXT i
        THIS.cEncrypted = cEncrypted_password
    ENDPROC
    * Процедура расшифровки пароля
    PROCEDURE Decrypt_It
        PARAMETERS cPassword
        cUnencrypted_password = " "
        FOR i = 1 TO LEN(cPassword)
            cLetter = SUBSTR(cPassword, i, 1)
            cUnencrypted_password = cUnencrypted_password + ;
            CHR((ASC(cLetter)-5)/2)
        NEXT i
        THIS.cUnencrypted = cUnencrypted_password
    ENDPROC
ENDDEFINE

```

В Visual Basic вместо одного универсального невизуального объекта используется достаточно большое число специализированных невизуальных объектов. Опишем те, которые наиболее часто используются при построении пользовательского приложения.

Объект **App** определяет или задает такие параметры, как заголовок приложения, путь к исполняемому файлу и его имя, имя файла контекстной справки и т. д. Этот объект имеет только свойства и не имеет ни событий, ни методов.

Объект **ClassModule** содержит свойства, которые управляют поведением класса так же, как программа может описывать свойства и методы класса. В приложении вы можете использовать этот объект как контейнер для программного кода, который описывает свойства и методы созданного вами класса. Объект **ClassModule** похож по своему поведению на форму, но без визуального интерфейса.

Объект **Clipboard** обеспечивает доступ к системному буферу обмена. Этот объект используется для работы с текстом и графикой, содержащейся в буфере обмена, и с его помощью можно обеспечить пользователю возможность копировать, вырезать и вставлять данные в приложении.

Объект **Collection** - коллекция - представляет собой набор объектов, к которому можно обращаться как к единому целому. Обычно в коллекцию помещают группу однотипных объектов (членов коллекции), а затем обращаются к ним по номеру, который объект занимает в коллекции. Но в принципе, в коллекции могут быть объединены и объекты разного типа. Размещение объектов в коллекции позволяет выполнять необходимые действия одновременно над всеми объектами, обращаясь только к одному объекту **Collection**. Объект создается точно так же, как и другие объекты, например: /p>

```
Dim oCollect1 As New Collection
```

Объекты в коллекцию могут быть добавлены с помощью метода **Add** и исключены из нее с помощью метода **Remove**.

Объекты OLE

Элементы управления OLE - этот дополнительный инструмент, который позволяет расширить функциональность разрабатываемых прикладных программ. Возможность использования элементов управления OLE в приложении появилась за счет поддержки технологии OLE 2.0 (Object Linking and Embedding) - стандартного интерфейса, разработанного Microsoft, для одновременного использования приложениями объектов в ОС Windows. Что дает эта технология разработчику прикладного программного обеспечения?

При разработке прикладной программы мы можем использовать объекты из других приложений. Например, разместить в форме текстовый документ Word for Windows. Текстовый документ станет OLE-объектом. Тогда текстовый процессор примет статус сервера OLE, а прикладная программа - клиента OLE. При обращении к объекту OLE все необходимые функции предоставляет сервер OLE. Он продолжает работать до тех пор, пока в других приложениях останется хотя бы один его объект. В Windows 95 появился новый тип серверов OLE - внутренний сервер OLE. Он предоставляет запрашивающим приложениям тот же интерфейс, что и внешний сервер OLE, однако отличается от него намного более высоким быстродействием, так как не требует отдельного пространства памяти для второго приложения. Другая возможность технологии OLE 2.0 заключается в использовании настраиваемых элементов управления ActiveX (OLE Custom Controls - OCX). Элементы управления ActiveX - это стандартный формат объектов OLE, допустимый для приложений, написанных и на других языках программирования. Их использование позволяет обеспечить пользовательскому приложению практически любую функциональность, реализовать которую встроенными средствами было бы невозможно или затруднительно.

Элемент управления **OLE Bound** создает встроенный OLE-объект, который связан с полем типа General таблицы Visual FoxPro. Встроенный OLE-объект не имеет собственного набора событий. Чаще всего в качестве встроенного OLE-объекта используется изображение, звук или видеоролик.

Элемент управления **OLE Container** позволяет включить элемент управления OLE в пользовательское приложение Visual FoxPro. Элемент управления OLE обычно может быть использован в виде файла с расширением OCX. Кроме того, в виде подобного элемента управления может использоваться файл документа Word for Windows или таблица Excel.

В Visual Basic функциональность обоих этих элементов управления обеспечивается одним объектом OLE.

Для работы с OLE-объектами можно использовать следующие свойства.

Control.AutoActivate [= nValue]

Определяет, как могут быть активизированы элементы управления OLE. Если параметр *nValue* равен 0, то элемент управления не будет автоматически активизирован при выборе пользователем. Вы можете активизировать элемент управления программно, используя метод DoVerb. Если *nValue* = 1, то прикладная программа, являющаяся для объекта сервером, будет активизирована после выбора объекта пользователем. Если *nValue* = 2 (по умолчанию), то объект OLE будет активизирован двойным щелчком мыши или нажатием клавиши **Enter** на этом объекте, если объект выбран. Если *nValue* = 2, то объект OLE будет активизирован двойным щелчком мыши на элементе управления или выбором этого объекта (автоматическая активизация).

Control.HostName [= cExpression]

Для некоторых приложений позволяет задать имя, которое будет использоваться в качестве заголовка окна при редактировании OLE-объекта.

Пользователи профессиональной версии Visual FoxPro получают набор дополнительных элементов управления ActiveX вместе с пакетом. Набор состоит из четырех файлов - MSCOMM32.OCX, MSMAPI32.OCX, MSOUTL32.OCX и PICCLIP32.OCX, - которые при инсталляции Visual FoxPro записываются в директорию SYSTEM операционной системы Windows. Эти элементы управления наряду с другими входят и в профессиональную версию Visual Basic. Что же таится в этих файлах?

MSCOMM32.OCX представляет **элементы управления для передачи данных** в приложении по последовательному интерфейсу (**Communications control**). Они обеспечивают следующие пути для управления передачей данных:

- Событийно управляемая передача данных позволяет с помощью события **OnComm** отслеживать начало передачи данных на компьютер (Carrier Detect) и необходимость передачи данных с компьютера (Request To Send). Событие OnComm также отслеживает ошибки соединения.

- Определять необходимость передачи данных можно и с помощью проверки значения свойства **CommEvent** после выполнения программой определенных действий, например, после получения сигнала **OK** от модема.

Для каждого последовательного порта, который используется для передачи данных, необходимо использовать отдельный элемент управления.

MSMAPI32.OCX - представляет набор элементов для создания в пользовательском приложении возможности передачи почтовых сообщений (**Microsoft MAPI Controls**). В файле содержатся два элемента управления:

- Элемент управления сеансом связи (**MAPI Session Control**) позволяет установить связь с требуемым получателем почтового сообщения.
- Элемент управления сообщениями (**MAPI Messages Control**) позволяет пользователю выполнять различные почтовые функции.

Эти элементы управления невидимы во время работы программы, к тому же они не имеют собственных событий. Для их использования необходимо применять соответствующие методы.

MSOUTL32.OCX - представляет элемент управления для создания иерархического списка (**Outline Control**), в котором каждый пункт может иметь подчиненный пункт, выводимый с отступом. Пользователь может сворачивать список, делая невидимыми подчиненные пункты, или, наоборот, раскрывать иерархию интересующих его пунктов. Такой тип списка используется в **Windows 95** для вывода списка файлов и директорий, а в **Visual FoxPro** для отображения элементов, входящих в проект.

PICCLP32.OCX - представляет элемент управления для вывода растровых изображений (**Picture Clip Controls**), который позволяет выбрать область в исходном изображении и затем вывести ее в форме. Этот элемент управления не видим в процессе работы программы. Он позволяет заменить множество используемых в программе файлов **BMP** или **ICO** одним файлом, содержащим все необходимые изображения. С помощью элемента управления для вывода изображений можно выбрать нужную область и вывести требуемую иконку или изображение, например, для каждой кнопки на панели инструментов. Это позволяет значительно эффективнее использовать память компьютера.

При установке **СУБД Access 7.0** вы получаете в свое распоряжение один дополнительный элемент управления - **календарь (Calendar Control)**.

Этот элемент управления находится в файле **MSACAL70.OCX** и может быть встроен в форму приложения для быстрого поиска даты. Календарь также поддерживает события, с помощью которых ваше приложение может автоматически выполнять какие-либо действия, связанные с той или иной датой.

Значительное количество дополнительных элементов управления поставляется вместе с **Visual Basic**, особенно с профессиональной версией этого средства разработки. Дадим им краткую характеристику.

В файле **COMCTL32.OCX** содержатся несколько элементов управления **Windows 95**.

Элемент управления **Listview** позволяет отображать списки, используя один из четырех способов:

- в виде значков (иконки);
- используя уменьшенные версии значков;
- в виде списка;
- в виде таблицы с дополнительным текстом для пояснения к каждому пункту.

Вы можете выстраивать пункты в колонках, используя заголовки и сочетая значки с текстом, а также устанавливать сортировку пунктов.

Элемент управления **ImageList** предназначен для хранения коллекции объектов **ListImage**, каждый из которых может быть идентифицирован по его номеру в списке или по ключу. Элемент управления **ImageList** не предназначен для самостоятельного использования, а может быть включен в любой другой элемент управления как хранилище изображений.

Элемент управления **TreeView** позволяет отображать иерархические списки специальных объектов **Node**, каждый из которых содержит метку и, возможно, изображение. Этот элемент управления можно эффективно использовать для отображения содержания каких-либо документов, списков файлов и папок и т. д. По своему назначению он аналогичен объекту **Outline Control**, но предоставляет более широкие возможности для программного управления сворачиванием и раскрытием иерархии пунктов, перемещением и выбором пунктов в списке, в том числе с использованием поиска первых символов по нажатию клавиш. Список выводится в окне, которое имеет полосы прокрутки. Элемент управления **TreeView** использует элемент управления **ImageList** для хранения значков, которые используются при выводе каждого объекта **Node** (пункта в

списке).

Элемент управления **ProgressBar** позволяет визуально контролировать длительность выполнения какого-то действия и представляет собой прямоугольник с движущейся в зависимости от скорости выполнения процесса полосой. В то же время этот объект оценивает только относительную длительность процесса без оценки каких-либо численных характеристик.

Элемент управления **StatusBar - строка состояния** - позволяет создать окно, обычно располагающееся внизу формы, в котором могут отображаться различные данные о текущем состоянии приложения. Строка состояния может быть разделена максимум на **16** объектов **Panel**, которые составляют коллекцию **Panels**. Каждый объект **Panel** может содержать определенный текст и (или) изображение. Дополнительно вы можете использовать одно из семи значений свойства **Style** для автоматического отображения таких данных, как дата, время, состояние триггерных клавиш на клавиатуре и т. п.

Элемент управления **Slider - движок** - это окно, содержащее движок со шкалой. Пользователь может перемещать движок, перетаскивая его, щелкая мышкой с разных сторон движка, или использовать клавиатуру. Основное назначение элемента управления **Slider** - выбор дискретных значений в каком-либо диапазоне. Вы можете располагать этот элемент управления как горизонтально, так и вертикально.

Элемент управления **TabStrip** предназначен для создания многостраничных форм с вкладками. По принципу действия он аналогичен объекту **PageFrame** в **Visual FoxPro**, но не является контейнером, то есть не может включать другие объекты. Доступ к страницам основан на том, что **TabStrip** содержит несколько объектов **Tab** из коллекции **Tabs**. Каждый объект **Tab** имеет свойства, ассоциированные с его текущим состоянием и внешним видом. Например, вы можете включить в **TabStrip** элемент управления **ImageList** и затем использовать изображения на конкретной вкладке, используя объект **Tab**. Заголовки вкладок **TabStrip** могут включать помимо текста также изображения.

Элемент управления **Toolbar - панель инструментов** - содержит коллекцию объектов **Button**, используемых для создания набора элементов управления в вашем приложении. Наличие панели инструментов является практически стандартным требованием к приложению **Windows** и обеспечивает пользователю быстрый доступ к часто выполняемым функциям. Изображения для кнопок, размещаемых на панели инструментов, хранятся в объекте **ImageList**. Элемент управления **Toolbar** имеет богатый набор свойств, методов и событий для эффективного управления панелью инструментов во время работы программы. Объекты **Button** поддерживают различные стили, допускающие создание группы кнопок, использование которых должно быть взаимоувязано, а стиль **PlaceHolder** допускает использование любых других элементов управления типа раскрывающегося списка.

Следующий элемент управления **Windows 95** находится в файле **RICHTX32.OCX** и обеспечивает возможность создания окна для работы с форматированным текстом - **RichTextBox**. В отличие от элемента управления **TextBox**, объект **RichTextBox** имеет ряд свойств, которые позволяют форматировать выделенный в этом объекте текст: изменять начертание шрифта, его цвет и использовать надстрочные и подстрочные индексы. Пользователь может также устанавливать вид выравнивания текста. Редактируемый текст может сохраняться как в **RTF**, так и в **ASCII** формате. За счет использования методов **LoadFile** и **SaveFile** элемент управления **RichTextBox** позволяет непосредственно считывать и записывать данные в файл. Поддерживается также открытие файла путем его перетаскивания, например из **Windows 95 Explorer**. Можно даже перетащить фрагмент текста из **Microsoft Word** и продолжить работу с ним в форме вашего приложения, включающей объект **RichTextBox**. Метод **SelPrint** позволяет распечатать весь текст или фрагмент данных на принтере.

В связи с тем, что объект **RichTextBox** является связанным объектом, вы можете ассоциировать его с элементом управления **Data** для отображения данных из полей примечаний. Элемент управления **RichTextBox** поддерживает большинство свойств, событий и методов, используемых стандартным элементом управления **TextBox**, но не имеет ограничения на размер выводимых с его помощью данных.

В файлах **COMDLG16.OCX** и **COMDLG32.OCX** содержатся 16- и 32-разрядные версии элемента управления **CommonDialog Control**, который обеспечивает набор стандартных диалоговых окон для таких операций, как открытие, сохранение и печать файлов, выбор цвета и шрифта. Этот элемент управления обеспечивает интерфейс между пользовательским приложением и функциями, размещенными в динамической библиотеке **Microsoft Windows COMMDLG.DLL**.

Вы можете легко включить в свою форму необходимые стандартные диалоги путем простого добавления в нее объекта **CommonDialog** и задания для него требуемых свойств. После включения в форму в режиме проектирования этот элемент управления изображается в виде значка. Соответствующее диалоговое окно появляется на экране путем выполнения соответствующего метода.

Следующие объекты **ActiveX** позволяют эффективно организовать работу с данными.

Файл **DBGRID32.OCX** содержит элемент управления **DBGrid**, который позволяет создать таблицу для просмотра и редактирования данных объекта **Recordset**. Возможность привязки объекта **DBGrid** к элементу управления **Data** позволяет автоматически заполнить заголовки

колонок на основании данных объекта **Recordset**. Элемент управления **DBGrid** содержит коллекцию **Columns** для формирования набора колонок. В каждой ячейке **DBGrid** может отображаться как текст, так и изображение, исключая связанные или внедренные объекты. Если текст слишком большой, чтобы разместиться по длине в ячейке, то он будет автоматически перенесен на следующую строку в той же ячейке. Объект **DBGrid** обеспечивает возможность программной ссылки на нужную ячейку. Свойства **Text** и **Value** объекта **Column** содержат значение текущей ячейки. Каждая колонка поддерживает свой шрифт, тип рамки, цвет и другие атрибуты, независимо от их установки в других колонках.

В файлах **GRID32.OCX** и **GRID16.OCX** содержатся соответственно 32- и 16- разрядные версии еще одного объекта **Grid** - **Microsoft Grid Control**. По своей функциональности он в основном аналогичен вышеописанному объекту - **DBGrid**.

Файл **DBLIST32.OCX** включает два элемента управления для вывода данных в виде списков - **DBList** и **DBCombo**. Эти элементы управления автоматически заполняют список из полей одного элемента управления **Data** и могут передавать данные из выбранного поля во второй элемент управления **Data**, например, для редактирования. Элементы управления **DBList** и **DBCombo** отличаются от стандартных элементов управления **ListBox** и **ComboBox** тем, что не требуют использования метода **AddItem** для формирования списка, а также поддерживают автоматический поиск без необходимости написания дополнительного кода.

В файлах **MSMASK16.OCX** и **MSMASK32.OCX** содержится элемент управления для ограничения вводимых данных по шаблону или их форматированного вывода на экран - **Masked Edit Control**. Этот элемент управления по использованию аналогичен стандартному элементу управления **TextBox**. Если вы опишете шаблон ввода, используя свойство **Mask**, каждое положение символа будет контролироваться по допустимому типу; станет также возможным использование заранее определенных символов, например, заключение междугородного телефонного кода в скобки: (812). При вводе данных курсор автоматически минует отображаемые символы шаблона, такие, например, как скобки в предыдущем примере. Попытка ввода символов, которые не будут соответствовать шаблону, сгенерирует событие **ValidationError**. Например, если задан шаблон "??###" и текущее отображаемое значение "A12.", то попытка ввести символ "B" перед "A" вызовет смещение символа "A" вправо, а так как в этом случае будет нарушен тип допустимого символа (число), то будет сгенерировано событие **ValidationError**.

В файлах **SPIN16.OCX** и **SPIN32.OCX** вы можете найти элемент управления **SpinButton** - **счетчик** - для ввода дискретных значений. Этот элемент управления по своим возможностям аналогичен рассмотренному ранее стандартному объекту **Visual FoxPro** - **Spinner**.

Альтернативой объекту **TabStrip** являются 16- и 32-разрядные версии элемента управления **SSTab**, который находится в файлах **TABCTL32.OCX** и **TABCTL16.OCX**. Этот элемент управления обеспечивает очень легкий путь создания многостраничных диалоговых окон в одной форме. Объект **SSTab** содержит страницы, каждая из которых может являться, в свою очередь, объектом-контейнером для любых других элементов управления. Используя соответствующие свойства, вы можете:

- задать число страниц;
- расположить вкладки более чем в одну строку;
- установить для вкладок соответствующие заголовки или изображения;
- выбрать требуемый стиль оформления;
- установить размеры каждой страницы.

Во время работы программы пользователь может перемещаться между страницами с помощью клавиш **Ctrl+Tab**.

Разнообразить пользовательский интерфейс вам поможет использование следующих объектов **ActiveX**.

В файлах **GAUGE16.OCX** и **GAUGE32.OCX** содержатся 16- и 32-разрядные версии элемента управления **Gauge**, широко известного среди программистов под жаргонным названием "термометр". Это шкальный индикатор, который может использоваться для визуального контроля длительности выполнения какого-то процесса аналогично уже описанному элементу управления **Progress-Bar**. От последнего **Gauge** отличается более точными показаниями, которые отслеживаются непрерывно.

С версией **Visual Basic** масштаба предприятия поставляется объект для доступа к внешним данным - **Microsoft Remote Data Control (MSRDC)**, который обеспечивает доступ к внешнему источнику данных с помощью технологии **ODBC**. Этот элемент управления обеспечивает интерфейс между объектами для доступа к внешним данным - **Remote Data Objects (RDO)** и элементами управления для отображения данных, расположенных в экранной форме. Используя элемент управления **MSRDC** вы можете:

- установить соединение с источником данных, используя свойства **MSRDC**;
- создать элемент управления для перемещения между записями;

- передавать данные из текущей строки в соответствующие элементы управления;
- получать и отображать данные о текущем положении указателя записи;
- обновлять данные в источнике после их изменения пользователем.

8. Подробнее об использовании объектов для доступа к внешним данным вы прочитаете в [главе](#)

В Visual FoxPro любой установленный на вашем компьютере объект ActiveX вы можете включить в форму, выбрав объект OLE Container Control, а затем в автоматически появляющемся диалоговом Insert Object окне выбрать опцию Insert Control. Из появляющегося списка можно выбрать соответствующий элемент управления, как это видно на рис. 5.6.

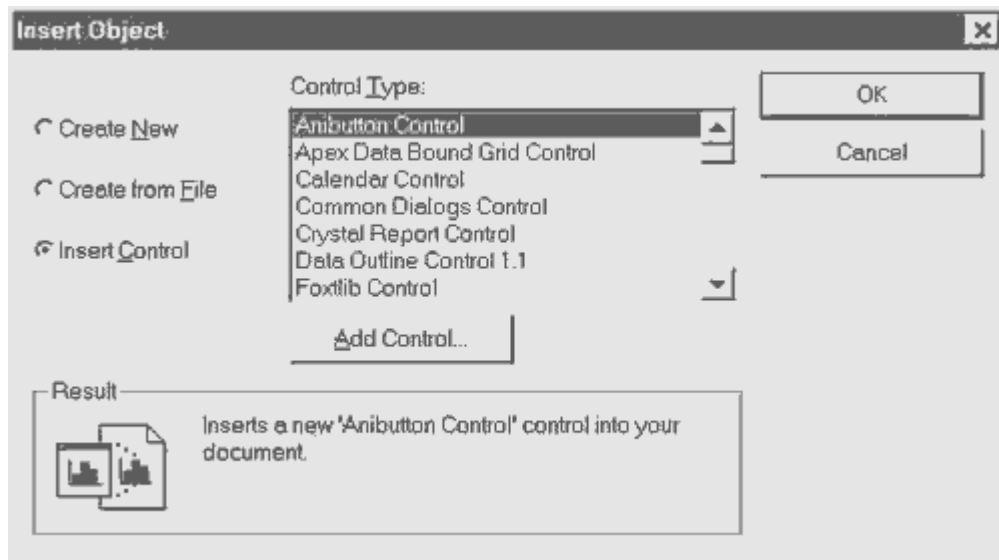


Рис. 5.6.

В Visual Basic подобную процедуру вы можете выполнить в любой момент работы над приложением. В меню *Tools* выберите команду *Custom Controls*, выберите из появляющегося списка нужный объект (рис. 5.7). Соответствующий объект вы затем можете поместить в проектируемую форму, выбрав его значок на панели инструментов *ToolBox*. На рис. 5.8 показана эта панель инструментов с описанными выше элементами управления.

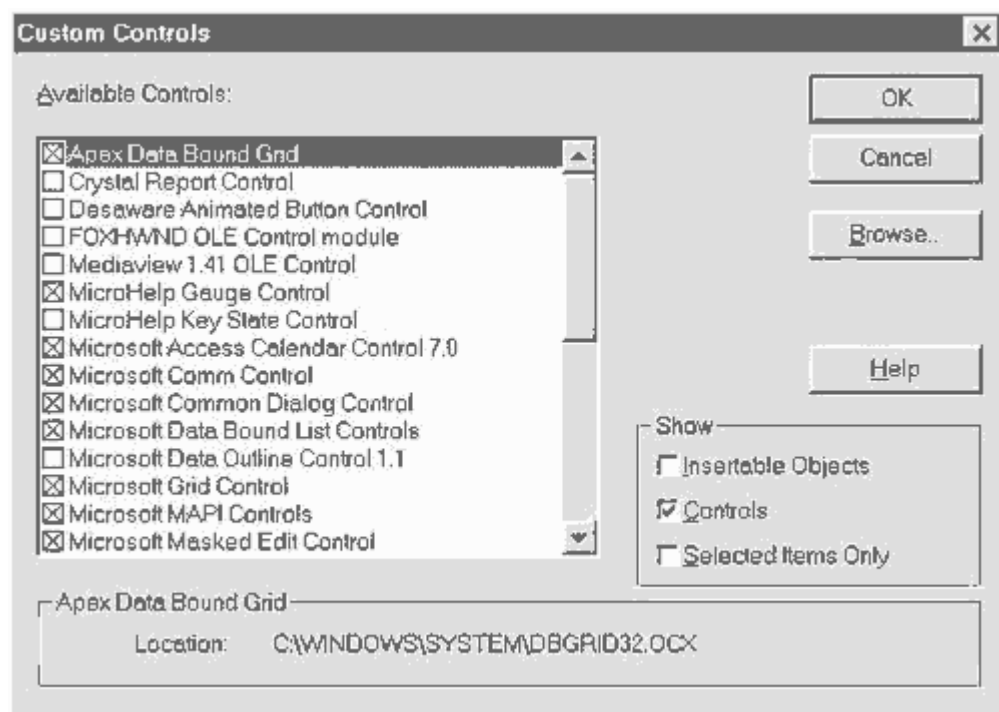


Рис. 5.7.

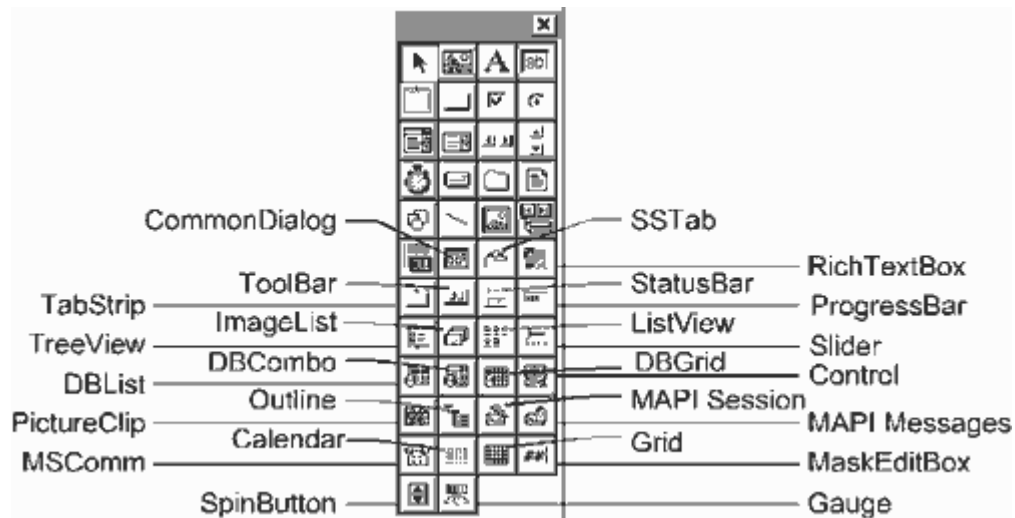


Рис. 5.8. Визуальные объекты и элементы управления в Visual Basic с подключенными компонентами ActiveX

Элементы управления OLE самого различного назначения в виде файлов OCX можно создать с помощью Control Development Kit в Microsoft Visual C++ или с помощью программного обеспечения других фирм. В этом случае вам придется вручную зарегистрировать те OLE-объекты, которые вы захотите использовать, с помощью программы REGSVR32.EXE. Эта программа находится в директории SAMPLES\OLE Visual FoxPro. При запуске программы REGSVR32 имя файла OCX надо указать в качестве параметра.

5.3. Управление событиями

Все многообразие свойств, которые имеют объекты, не выглядело бы столь привлекательным, если бы программист не мог присваивать их или менять в зависимости от условий работы пользовательского приложения. Расширить возможности управления объектами за рамки определений, устанавливаемых при их создании, и призваны события, генерируемые для объектов приложения.

В этом параграфе мы познакомимся и научимся заставлять программу реагировать на события, вызываемые действиями пользователя.

События генерируются системой при выполнении каких-либо действий пользователя, например щелчком мыши на объекте или нажатием клавиши при вводе данных, а также могут быть вызваны принудительно из программы или сгенерированы операционной системой.

При возникновении события приложение выполняет определенные для этого события действия. Например, если в форме пользователь щелкает мышкой на каком-либо объекте, то объект активизируется. Программист всегда может либо переопределить эти действия, либо дополнить их, написав соответствующий код в процедуре для данного события.

При работе с объектом-контейнером включенные в него объекты имеют свои события, и если, например, пользователь щелкает мышкой на кнопке управления в форме, то выполняется событие Click и обрабатывается соответствующая процедура для данной кнопки, а не для формы. При управлении объектами-контейнерами следует придерживаться следующих правил:

- Объекты-контейнеры не выполняют код для событий, если действия для этих событий предусмотрены во включенном в него объекте.
- Если для данного события реакция элемента управления не описана, приложение ищет код для данного события в более высоком уровне классовой иерархии. Как только такой код будет найден, он будет выполнен и дальнейший поиск прекратится. В этом случае при разработке приложения в Visual FoxPro не забудьте про оператор ":", с помощью которого можно выполнить код более высокого уровня.

Если рассматривать события с точки зрения управления каким-либо объектом, то надо говорить о последовательности событий, которые происходят с момента его создания до уничтожения. Последовательность возникновения событий жестко определена. Разговор о событиях начнем с трех событий, которые распространяются на наибольшее число объектов.

PROCEDURE *Object.Init*
[LPARAMETERS *Param1, Param2,...*]

Это первое событие в последовательности событий при создании объекта. Обычно событие **Init** используется для получения параметров из вызывающей процедуры *Param1*, *Param2*, и выполнения установочных действий типа считывания нужных данных и т. п.

Сказанное выше правильно для всех рассматриваемых в книге средств разработки, но нет полной радости у программиста. В **Visual FoxPro** упомянутое выше событие называется **Init**, а в **Visual Basic** примерно аналогичное событие называется **Initialize**. При этом процедура для описания действий, которые должны произойти при наступлении события, в **Visual Basic** имеет следующую структуру:

```
Private Sub ObjectName_EventName(<<Параметры>>)
    <<Описание переменных и констант>>
    <<Операторы>>
End Sub
```

Имя события указывается после имени объекта, которому оно соответствует, и отделяется знаком подчеркивания. Тогда для события заголовок процедуры должен быть написан так:

```
Private Sub Object_Initialize()
```

В **Visual Basic** это событие может относиться только к форме и объекту **ClassModule**.

Событие **Init** всегда выполняется сначала для объекта, включенного в объект-контейнер, и лишь затем для самого объекта-контейнера. Если событие **Init** возвращает значение **.F.**, то объект не будет создан. Например, если мы создадим форму, в которой собираемся работать с данными из какой-либо таблицы, а файла таблицы нет на диске, то следующий пример, написанный на **Visual FoxPro**, показывает, как в этом случае можно избежать загрузки формы.

```
oFrm = CREATEOBJECT("frmLook") && Создаем объект формы
IF type("oFrm") != "O"
    RETURN
ENDIF
oFrm.Show && Выводим ее на экран
READ EVENTS && Включаем состояние ожидания
* Описываем класс формы
DEFINE CLASS frmLook AS Form
    ScaleMode = 3 && Координаты в пикселях
    Width = 320 && Ширина
    Height = 200 && Высота
    Closable = .F. && Закрытие только кнопкой
* Добавляем объект Grid, с помощью которого будем выводить данные
ADD OBJECT grdCustomer AS Grid WITH ;
    RecordSource = "Customer", ;
    RecordSourceType = 1
* Добавляем кнопку для выхода из формы
ADD OBJECT cmdExit AS CommandButton WITH ;
    Caption = "Выход", ;
    Top = 220, ;
    Left = 100
* Описываем действия при создании объекта Grid - проверка файла
PROCEDURE grdCustomer.Init
    IF !FILE("CUSTOMER.DBF")
        =MESSAGEBOX("Файл данных не найден!")
    RETURN .F.
    ELSE
        IF USED("CUSTOMER")
            SELECT CUSTOMER
        ELSE
            USE CUSTOMER IN 0
        ENDIF
    ENDIF
ENDPROC
* Описываем действия при щелчке мышкой на кнопке
PROCEDURE cmdExit.Click
    CLEAR EVENT && Отменяем состояние ожидания
ENDPROC
ENDDEFINE
```

Последним событием при существовании объекта является событие **Destroy**. Это событие обычно используется для выполнения завершающих действий при работе с объектом, например, запись измененных данных и т. п.

Аналогично событию **Init** и его примерному аналогу в Visual Basic - **Initialize**, для события **Destroy** в Visual Basic существует примерный аналог - событие **Terminate**.

Одним из наиболее часто используемых событий для управления работой программы является событие **Click**. Это событие происходит, когда пользователь:

- Щелкает левой кнопкой мыши на элементах управления **CheckBox**, **CommandButton** или **Optionbutton**, на пустом месте формы или в области ввода данных элемента управления **Spinner**.
- Выбирает пункт в списке элементов управления **Combobox** или **ListBox**.
- Нажимает клавишу **Space** (Пробел), когда активен элемент управления **CheckBox**, **CommandButton** или **OptionButton**.
- Нажимает клавишу **Enter**, когда один из элементов управления **CommandButton** имеет свойство **Default**, равное **.T.**
- Нажимает "горячую клавишу", определенную для какого-либо элемента управления.

Событие **Click** наступает также, когда выполняются следующие действия в программе:

- Значение свойства **Value** элементов управления **CommandButton** или **Optionbutton** устанавливается в значение **.T.** или **1** для **OptionButton**.
- Изменяется значение свойства **Value** элемента управления **CheckBox**.
- Выполняется команда **MOUSE**.

Событие **DbClick** происходит, когда пользователь дважды быстро нажимает и отпускает левую кнопку мыши на каком-либо объекте. Для объектов **ComboBox** и **ListBox** это событие происходит также при выборе какого-либо пункта из списка и нажатии при этом клавиши **Enter**. При привязке каких-либо действий к событию **DbClick** следует помнить, что пользователи не всегда могут четко выполнить данное действие и двойной щелчок может быть воспринят системой как два одинарных щелчка, выполненных с небольшим интервалом.

При выборе объекта, то есть при установке на него курсора, для этого объекта выполняется событие **GotFocus**. Это же событие выполняется при использовании в программе метода **SetFocus**. Следует учесть, что стать активным объект может, только если его свойства **Enabled** и **Visible** равны **.T.**

Когда пользователь выбирает другой объект в форме для дальнейшей работы, текущий объект деактивируется и для него происходит событие **LostFocus**.

Следующее событие происходит для каждого объекта, размещенного на странице формы (**Page**), когда эта страница активизируется или деактивируется.

PROCEDURE *Object.UIEnable*
LPARAMETERS *IEnabled*

Параметр *IEnabled* может иметь значение **.T.** или **.F.**, что будет свидетельствовать о, соответственно, активизации или деактивизации страницы, на которой расположен объект. Таким образом, это событие позволяет предпринимать какие-либо действия с элементом управления, например, обновлять данные при переходе пользователя на новую страницу в форме.

Во время редактирования данных при каждом изменении пользователем значения свойства **Value** элемента управления происходит событие **InteractiveChange** для редактируемого элемента. Если изменение значения произведено из программы, для элемента управления происходит событие **ProgrammaticChange**.

В Visual Basic эти функции приходится выполнять одному событию - **Change**. Его синтаксис:

Private Sub *Object_Change*([*Index* As Integer])

Параметр *Index* позволяет получить номер объекта, если он включен в массив элементов управления.

Для элементов управления, предусматривающих ввод данных, при нажатии и отпускании клавиши клавиатуры происходит событие **KeyPress**. Его синтаксис в Visual FoxPro выглядит так:

PROCEDURE *Object.KeyPress*
LPARAMETERS *nKeyCode*, *nShiftAltCtrl*

Параметр *nKeyCode* возвращает код нажатой клавиши, совпадающий с кодом, возвращаемым функцией **INKEY()**. Параметр *nShiftAltCtrl* возвращает признак нажатия дополнительной клавиши. Клавиша **Shift** имеет код 1, **Ctrl** - 2, **Alt** - 4. Таким образом, если этот параметр имеет значение 1 - при вводе удерживалась клавиша **Shift**, если 6 - удерживались клавиши **Ctrl** и **Alt**. В **Visual Basic** это событие имеет следующий синтаксис:

```
Private Sub Form_KeyPress(Keyascii As Integer)
```

Параметр *Keyascii* возвращает числовой **ASCII**-код, который передается по ссылке. Следующий пример позволяет преобразовывать все вводимые в текстовое поле символы в верхний регистр:

```
Private Sub Text1_KeyPress (KeyAscii As Integer)
    Char = Chr(KeyAscii)
    KeyAscii = Asc(UCase(Char))
End Sub
```

Помимо этого события в **Visual Basic** есть еще два события, которые позволяют отслеживать работу пользователя с клавиатурой. Это событие **KeyDown** - срабатывает при нажатии клавиши и **KeyUp** - при ее отпускании. Эти события могут обрабатываться совершенно аналогично событию **KeyPress** в **Visual FoxPro**, то есть с их помощью можно отслеживать нажатие и комбинаций клавиш.

Для большинства элементов управления в **Visual FoxPro** перед их активизацией возможность перехода в этот элемент управления проверяется с помощью события **When**. Если это событие возвращает **.T.** (по умолчанию), то управление передается на этот элемент управления и возникает следующее событие **GotFocus**. Если событие **When** возвращает **.F.**, то этот элемент управления не активизируется. Для объекта типа **List** событие **When** возникает всякий раз, когда пользователь перемещается между пунктами в списке.

Перед тем как передать управление следующему объекту, для текущего элемента управления в **Visual FoxPro** выполняется событие **Valid**. Если это событие возвращает значение **.T.**, то данный элемент управления деактивизируется. Если **.F.**, то элемент управления остается активным. Таким образом, событие **Valid** можно использовать для проверки допустимости значения, выбранного для данного элемента управления.

В связи с тем, что основным объектом пользовательского интерфейса является форма, уделим теперь внимание характерным для этого объекта событиям.

Событие **Activate** происходит, когда пользователь активизирует ранее не активную форму (щелкает мышкой на форме или входящем в нее элементе управления) или когда вызывается метод **Show**.

Событие **Deactivate** происходит, когда форма перестает быть активна, то есть управление передается другому объекту в том же приложении.

Следует учитывать, что ни событие **Activate**, ни **Deactivate** не происходят при передаче управления в другое приложение. Событие **Deactivate** также не происходит при стирании формы из памяти.

При создании формы (ее загрузки в память) перед событиями **Activate** и **SetFocus** происходит событие **Load**.

При стирании формы из памяти происходит событие **Unload**.

Событие **Paint** происходит, когда форма перерисовывается. Форма перерисовывается после перемещения или увеличения, а также, если было сдвинуто окно, ранее его закрывающее.

Использование события **Paint** для некоторых задач может вызывать каскад последующих событий. Если у вас нет веских причин, избегайте использовать событие **Paint** в следующих случаях:

- Перемещение или изменение размеров формы или элементов управления.
- Изменение любых переменных, которые воздействуют на размер или вид, типа установки свойства **BackColor**.
- Вызов метода **Refresh**.

При программировании реакции пользовательского приложения на происходящие события очень важно знать последовательность их возникновения. Рассмотрим эти последовательности для основных объектов **Visual FoxPro**. В **Visual Basic**, как вы, наверное, уже заметили, событий, на которые может реагировать приложение, значительно меньше и поэтому последовательность их возникновения для отдельного объекта не имеет такого значения.

Для набора форм и входящих в него отдельных форм (пусть для примера в набор входят две формы) будет выполняться следующая последовательность событий при его запуске:

```

FormSet.DataEnvironment.BeforeOpenTables (если свойство AutoOpenTables = .T.)
FormSet.Load
FormSet.Form1.Load
FormSet.Form2.Load
FormSet.DataEnvironment.<<Объекты Data Environment>>.Init
FormSet.DataEnvironment.Init
FormSet.Form1.<<Объекты в Form1>>.Init
FormSet.Form1.Init
FormSet.Form2.<<Объекты в Form2>>.Init
FormSet.Form2.Init
FormSet.Init
FormSet.Activate
FormSet.Form2.Activate
FormSet.Form2.Deactivate
FormSet.Activate
FormSet.Form1.Activate
FormSet.Form1.<<Первый элемент управления>>.When
FormSet.Form1.GotFocus
FormSet.Form1.<<Первый элемент управления>>.GotFocus

```

При завершении работы с набором форм выполняется следующая последовательность событий:

```

FormSet.Activate
FormSet.Form2.Destroy
FormSet.Form2.<<Объекты в Form2>>.Destroy
FormSet.Form2.Unload
FormSet.Form1.Destroy
FormSet.Form1.<<Объекты в Form1>>.Destroy
FormSet.Form1.Unload
FormSet.Unload
FormSet.DataEnvironment.AfterCloseTables
FormSet.DataEnvironment.Destroy
FormSet.DataEnvironment.<<Объекты Data Environment>>.Destroy

```

При активизации объекта Grid пользователем происходит следующая последовательность событий:

```

Form.Grid.When
Form.GotFocus
Form.Grid.AfterRowColChange

```

Последнее событие **AfterRowColChange** возвращает номер колонки, в которой находится курсор, то есть которая является активной. Это событие сопровождает и каждое перемещение пользователя по клеткам Grid, и таким образом, любой переход курсора в другую клетку сопровождается двумя событиями:

```

Form.Grid.BeforeRowColChange
Form.Grid.AfterRowColChange

```

Событие **BeforeRowColChange** возвращает номер колонки, в которой находился курсор до перехода в новую клетку.

Если пользователь пометил запись для удаления или в программе выполнялась команда **DELETE**, для объекта Grid генерируется событие **Deleted**, которое возвращает номер помеченной для удаления записи.

При прокрутке данных с помощью линейки прокрутки каждое нажатие на кнопку прокрутки или перемещение движка вызывает выполнение события **Scrolled**, которое возвращает следующий код для идентификации действий пользователя:

- 0 - нажата кнопка прокрутки вверх.
- 1 - нажата кнопка прокрутки вниз.
- 2 - пользователь щелкнул мышкой на вертикальной линейке прокрутки сверху от движка.
- 3 - пользователь щелкнул мышкой на вертикальной линейке прокрутки снизу от движка.

- 4 - нажата кнопка прокрутки влево.
- 5 - нажата кнопка прокрутки вправо.
- 6 - пользователь щелкнул мышкой на горизонтальной линейке прокрутки слева от движка.
- 7 - пользователь щелкнул мышкой на горизонтальной линейке прокрутки справа от движка.

При переходе к другому элементу управления из Grid происходит следующая последовательность событий:

```
Form.Grid.Valid
Form.<<Выбранный элемент управления>>.When
Form.Grid.BeforeRowColChange
```

Комбинированный список имеет несколько специфических событий. Если проследить последовательность действий пользователя при активизации комбинированного списка, нажатии на раскрывающую стрелку, нажатии на кнопку прокрутки списка вниз, а затем вверх и при выборе пункта, то мы получим следующую последовательность событий:

```
Form1.Combo1.When
Form1.GotFocus
Form1.Combo1.GotFocus
Form1.Combo1.MouseDown,1,0,108,48
Form1.Combo1.DropDown && Нажатие на раскрывающую стрелку
Form1.Combo1.MouseUp,1,0,108,48
Form1.Combo1.MouseDown,1,0,88,117
Form1.Combo1.MouseUp,1,0,88,117
Form1.Combo1.DownClick && Нажатие на кнопку прокрутки вниз
Form1.Combo1.MouseDown,1,0,92,11
Form1.Combo1.MouseUp,1,0,92,11
Form1.Combo1.UpClick <|>&& Нажатие на кнопку прокрутки вверх
Form1.Combo1.MouseDown,1,0,79,11
Form1.Combo1.MouseUp,1,0,29,27
Form1.Combo1.InteractiveChange && Изменение значения в списке
Form1.Combo1.Click
Form1.Combo1.Valid
Form1.Combo1.When
Form1.Combo1.Valid
Form1.<<Следующий элемент управления>>.When
Form1.Combo1.LostFocus
```

Для текстового поля при вводе в него данных с помощью возникающих событий мы можем отследить ввод каждого символа. Поэтому рассмотрим последовательность событий для этого элемента управления. Если текстовое поле Text1 будет размещено в форме Form1, и в текстовое поле мы перейдем с помощью мыши, и будем вводить три символа Rus, а затем нажмем клавишу Enter, то для этого текстового поля произойдет следующая последовательность событий:

```
Form1.Text1.When      && Проверка доступности
Form1.Text1.Gotfocus
Form1.Text1.MouseDown, 1, 0, 25, 76
Form1.Text1.MouseUp, 1, 0, 25, 76
Form1.Text1.Click
Form1.Text1.KeyPress, 82,1 && Нажимаем клавишу R
Form1.Text1.InteractiveChange
Form1.Text1.KeyPress, 117,0 && Нажимаем клавишу u
Form1.Text1.InteractiveChange
Form1.Text1.KeyPress, 115,0 && Нажимаем клавишу s
Form1.Text1.InteractiveChange
Form1.Text1.KeyPress, 13,0 && Нажимаем клавишу Enter
Form1.Text1.Valid
Form1.Text1.LostFocus
```

5.4. Использование методов

В этом параграфе мы остановимся на методах, которые можно использовать при работе с большинством объектов.

Если мы хотим в **Visual FoxPro** создать какой-либо объект в объекте-контейнере, например в форме, то в этом случае мы должны не создавать его с помощью функции **CREATEOBJECT()**, а добавлять его в объект-контейнер, используя метод

**Object.AddObject(cName, cClass [, cOLEClass]
[, aInit1, aInit2 ...])**

С помощью параметра **cName** мы задаем имя добавляемого объекта, а параметром **cClass** указываем класс, на основе которого будет создан объект в объекте-контейнере, указанном в **Object**. Определение класса, на основе которого создается объект, должно быть доступно, то есть если это не базовый класс, мы должны заранее открыть соответствующую библиотеку классов командой **SET CLASSLIB**. Если мы добавляем OLE-объект, то класс, на котором он будет основан, можно указать с помощью параметра **cOLEClass**. С помощью **aInit1, aInit2...** можно передать параметры в событие **Init** создаваемого объекта.

В **Visual Basic** неким аналогом этого метода может являться метод **Add**, который позволяет добавлять объект в коллекцию.

Для удаления объекта из его контейнера в **Visual FoxPro** используется метод **RemoveObject**, а для удаления объекта из коллекции в **Visual Basic** - метод **Remove**.

Если объект использует данные, которые могут обновляться во время работы программы, использовать их последний вариант поможет метод

Object.Refresh

который позволяет немедленно перерисовать форму или указанный объект. Причем, когда перерисовывается форма, автоматически обновляются все содержащиеся в ней элементы управления, когда перерисовывается страничный блок **PageFrame**, автоматически обновляется только активная страница **Page**.

Control.SetFocus

Позволяет активизировать указанный элемент управления.

[Object.]ZOrder([nValue])

Позволяет установить графический уровень отображения указанного объекта. Параметр **nValue** может принимать либо значение 0, либо 2. Если **nValue** равен 0 (по умолчанию), то объект выводится на переднем плане, если 2 - на заднем. Задний план обычно используется для отображения результатов работы графических методов (рисование линий, вывод изображений и т. д.), а передний - для отображения объектов. Объекты, отображаемые на переднем плане, перекрывают изображение заднего плана.

**Object.Move(nLeft [, nTop [, nWidth
[, nHeight]])**

Позволяет переместить объект в нужную точку, которая задается новыми координатами левой **nLeft** и верхней **nTop** границами объекта. При необходимости можно задать для объекта новую ширину **nWidth** и высоту **nHeight**. Обязательным является параметр **nLeft**, остальные должны указываться без пропусков, то есть если вы задаете ширину объекта, то должны использовать в методе все параметры.

Если перемещаемый объект входит в объект-контейнер, то новые координаты указываются относительно левого верхнего угла объекта-контейнера, которые равны 0, 0. Например, если вы хотите переместить форму, размеры которой установлены в фокселях, на пятую строку и двадцатую колонку экрана, задайте следующий код:

**ThisForm.Move(20, 5)
Object.Hide**

Скрывает форму путем присвоения свойству **Visible** значения **.F.** Элементы управления в скрытой форме, естественно, становятся недоступными для пользователя, но доступны для воздействия на них с помощью программы.

Object.Show([nStyle])

Выводит на экран форму и, если указан параметр *nStyle*, определяет способ ее вывода. Если параметр *nStyle* равен 1 (0 в Visual Basic), то форма является независимой и выполняется код, следующий после строчки, задающей выполнение метода **Show**. По умолчанию параметр *nStyle* равен 2 (1 в Visual Basic), и в этом случае программа ждет, когда пользователь завершит работу с формой или когда форма будет выгружена из памяти. До этого момента переход в другую форму или меню невозможен. Если параметр *nStyle* не указывается, то способ вывода формы определяется значением свойства **WindowType**.

Для OLE-объекта можно использовать метод

Object.DoVerb[Verb]

Выполняет команду для указанного объекта OLE. Параметр *Verb* - это одна из команд, поддерживаемая всеми объектами и предназначенная для выполнения в объекте-контейнере. Если параметр не указывается, выполняется команда, принятая по умолчанию для данного объекта. Параметр может также указывать номер в массиве свойства **ObjectVerbs**.

Следующий список содержит возможные номера для стандартных команд:

Значение	Действие
0	Действие для объекта по умолчанию
-1	Активизирует объект для редактирования
-2	Открывает объект в отдельном окне приложения
-3	Для включенного объекта скрывает приложение, создавшее объект
-4	Активизирует объект при условии, что его активизация поддерживается. При этом выводятся средства пользовательского интерфейса для редактирования объекта на месте (on-place editing)
-5	Создает окно для объекта и загружает необходимые для редактирования этого объекта средства, когда пользователь щелчком мыши активизирует объект-контейнер OLE
-6	Отменяет все выполненные в процессе модернизации изменения (выполняет действие UNDO) для объектов, открытых для редактирования

Если вы установите для свойства **AutoActivate** значение 2, то объект-контейнер OLE автоматически активизирует объект, когда пользователь сделает двойной щелчок мышью на элементе управления.

Использование имени команды (**EDIT**, **OPEN**, **PLAY** и т. д.) в параметре *Verb* всегда обеспечивает значительно большую скорость работы, чем использование номера команды.

Упомянем еще несколько специфических для Visual FoxPro методов, которые часто используются при создании приложения.

Container.SetAll(cProperty, Value [, cClass])

Позволяет в указанном объекте-контейнере присвоить значение *Value* для свойства *cProperty* всем размещенным в нем элементам управления или относящимся к классу *cClass* (не базовому классу Visual FoxPro!).

Для Grid могут быть выполнены следующие специфические методы:

Grid.ActivateCell(nRow, nColumn)

Активизирует клетку с указанными координатами номера строки *nRow* и колонки *nColumn*.

Grid.AddColumn(nIndex)

Позволяет добавить колонку в Grid. Колонка с номером *nIndex* сдвигается вправо, и ее номер увеличивается на 1.

Grid.DeleteColumn([nIndex])

Позволяет убрать текущую или указанную посредством номера *nIndex* колонку из Grid. Для прокрутки данных можно использовать метод

Grid.DoScroll(*nDirection*)

Для выполнения нужного действия аргумент *nDirection* может принимать следующие значения:

- 0 - прокрутка на строку вверх.
- 1 - прокрутка на строку вниз.
- 2 - прокрутка на страницу вверх.
- 3 - прокрутка на страницу вниз.
- 4 - прокрутка на колонку влево.
- 5 - прокрутка на колонку вправо.
- 6 - прокрутка на страницу влево.
- 7 - прокрутка на страницу вправо.

Глава 6

Создание базы данных

6.1. Visual FoxPro

Создание и модернизация структуры базы данных

Использование словаря данных

Создание и модернизация структуры таблиц

6.2. Access

6.3. Visual Basic

6.4. MS SQL Server

Планирование процесса наращивания

Описывая практическую технологию создания БД, мы коснемся основополагающих принципов работы с данными, используемых в средствах разработки корпорации Microsoft. Поэтому в данной главе мы разделим излагаемый материал, во избежание путаницы в головах наших читателей, по описываемым в книге средствам разработки.

6.1. Visual FoxPro

В Visual FoxPro базы данных имеют собственную структуру организации данных и предоставляют дополнительные возможности разработчикам. В базе данных вы можете использовать расширенное представление данных на уровне таблиц, например, правила на уровне полей и записей, значения полей по умолчанию, триггеры. Здесь же вы можете хранить процедуры и устанавливать постоянные отношения между таблицами. Базы данных можно использовать для обеспечения доступа к внешним источникам данных и для создания представлений локальных и внешних таблиц.

В этом параграфе мы рассмотрим:

- визуальные методы создания БД и ее компонентов;
- использование основных команд FoxPro для программного создания компонентов БД;
- организацию связей между таблицами и использование индексов.

Создание и модернизация структуры базы данных

После того, как вы спроектировали базу данных, ее можно создать в интерактивном режиме, используя диалоговые средства Visual FoxPro или команду **CREATE DATABASE**. Если вы разрабатываете пользовательское приложение, обязательно создавайте базу данных, используя Project Manager.

В Project Manager выберите вкладку Data, затем в списке пункт Databases. Нажмите кнопку New. Откроется Конструктор БД, который показан на рис. 6.1. На этом же рисунке вы найдете необходимые пояснения для работы с ним.

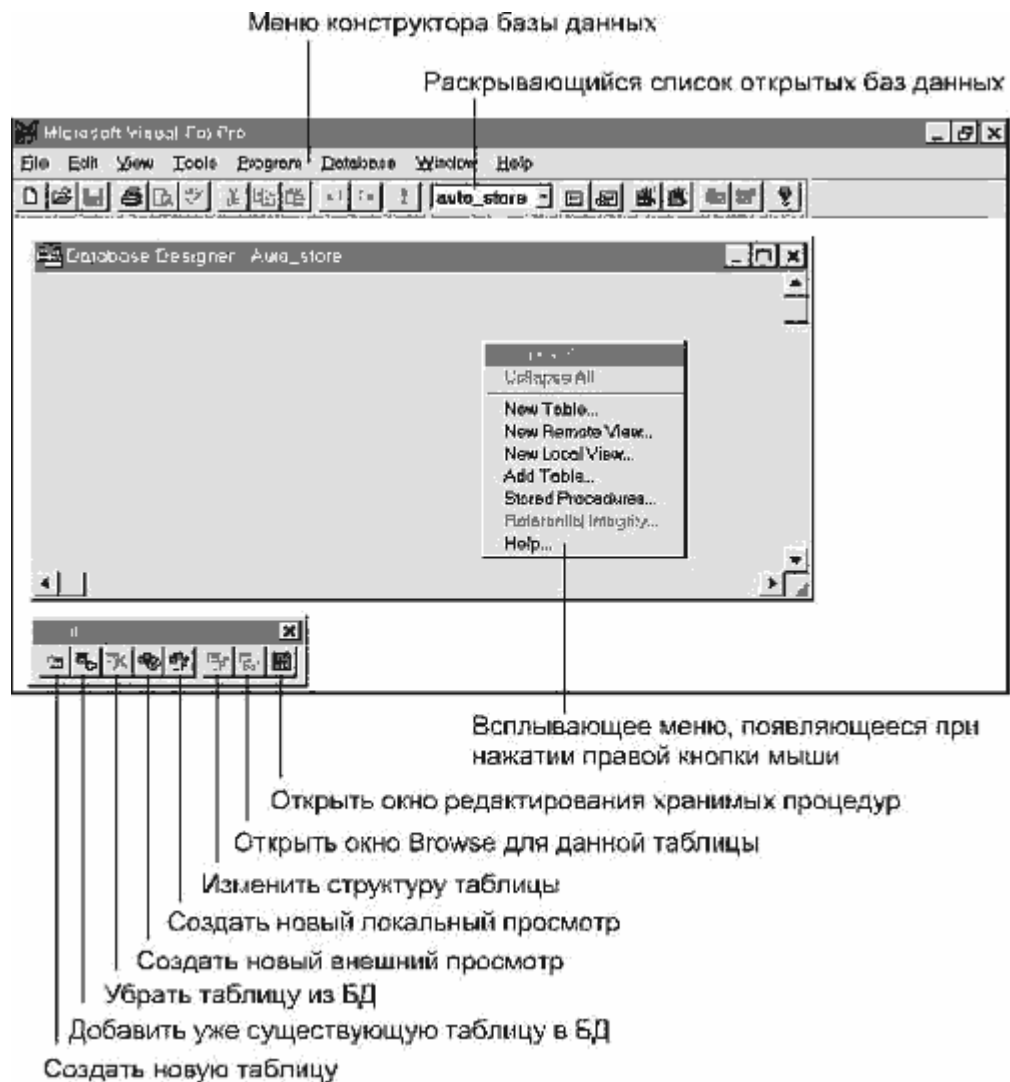


Рис. 6.1. Окно и панель инструментов Конструктора БД в Visual FoxPro

Вновь созданная база данных пуста и не содержит никаких таблиц и других объектов. В табл. 6.1 приведены команды и функции, которые вы можете использовать для программного управления базами данных и их объектами.

Таблица 6.1. Команды и функции, управляющие базами данных и их объектами

ADATABASE()	CREATE VIEW	MODIFY CONNECTION
ADBOBJECTS()	DBC()	MODIFY DATABASE
ADD TABLE	DBGETPROP()	MODIFY PROCEDURE
ALTER TABLE	DBSETPROP()	MODIFY STRUCTURE
APPEND PROCEDURES	DELETE CONNECTION	MODIFY VIEW
CLOSE DATABASE	DELETE DATABASE	OPEN DATABASE
COPY PROCEDURES	DELETE VIEW	PACK DATABASE
CREATE CONNECTION	DISPLAY DATABASE	REMOVE TABLE
CREATE DATABASE	INDBC()	SET DATABASE
CREATE SQL VIEW	LIST DATABASE	VALIDATE

DATABASE

CREATE TABLE

При создании БД не забудьте, что важной особенностью Visual FoxPro является то обстоятельство, что каждая таблица может существовать в одном из двух состояний: либо в виде свободной таблицы, представляющей собой файл DBF, не ассоциированный ни с одной из баз данных, либо в виде таблицы базы данных - файла DBF, включенного в какую-либо БД. При этом если таблица включена в БД, то она не может быть включена ни в какую другую БД. Для таблиц, ассоциированных с базой данных, можно установить ряд свойств, которыми не могут обладать свободные таблицы. Это такие свойства, как триггеры, правила на уровне полей и записей и постоянные отношения между таблицами.

Вы можете включить таблицы в базу данных, создав их внутри открытой базы данных или добавив туда уже существующие таблицы.

Как добавить существующую таблицу в базу данных? В Project Manager выберите пункт Tables из вкладки All или Data, затем выполните команду Add. Программным путем то же самое можно сделать, выполнив следующие команды:

```
OPEN DATABASE auto_store && Открывается база данных && AUTO_STORE
```

```
ADD TABLE salesman && В нее добавляется таблица SALESMAN
```

Чтобы сделать таблицу частью базы данных, необходимо явно включить ее в базу данных. Изменение структуры свободной таблицы не вносит ее автоматически в базу данных, даже если сделать это при открытой базе данных с помощью команды **MODIFY STRUCTURE**.

Не забудьте, что вы можете ассоциировать таблицу только с одной базой данных. Если данные из какой-то таблицы необходимо использовать в нескольких БД, для исключения дублирования информации наиболее простой путь - не включать эту таблицу ни в какую БД, а оставить ее свободной. Данные из существующего файла DBF можно использовать, не включая его в базу данных. В то же время не представляет труда использовать информацию и из таблицы, расположенной в другой БД.

Как получить доступ к таблице из другой базы данных? Создайте в базе данных представление, включающее нужную таблицу. Или получите доступ к таблице с помощью команды **USE** и разделительного символа "!". Символ "!" надо использовать для ссылки на таблицу из другой, не текущей, базы данных. Например, если вы хотите просмотреть таблицу Salesman из базы данных Auto_Store, выполните следующие команды:

```
USE Auto_Store!Salesman
BROWSE
```

В предыдущем примере база данных Auto_Store открывается автоматически при выполнении команды **USE**, однако Visual FoxPro не устанавливает Auto_Store в качестве текущей базы данных. Открытая таким способом база данных автоматически закрывается, когда вы закрываете таблицу, если база данных не была явно открыта до этого.

Когда вы добавляете таблицу в базу данных, Visual FoxPro изменяет заголовок файла таблицы, включая туда информацию о пути и имени файла базы данных. Этот путь и имя файла базы данных называются обратной связью, поскольку эта информация связывает таблицу с владеющей ею базой данных. Процесс удаления таблицы из базы данных не только удаляет ссылку на таблицу и ассоциированную с ней информацию из словаря БД Visual FoxPro, но и изменяет обратную связь, чтобы отметить таблицу как свободную.

Вы можете удалить таблицу из базы данных интерактивно или с помощью команды **REMOVE TABLE**. При удалении таблицы из базы данных вы можете и физически удалить файл таблицы с диска. В Project Manager выберите имя таблицы и нажмите кнопку Remove. Если вы работаете в Конструкторе баз данных, выделите имя таблицы и выберите команду **Remove** из меню **Database**.

Последовательность действий по удалению таблицы Salesman из БД Auto_Store программным путем показана в следующем примере:

```
OPEN DATABASE Auto_Store
REMOVE TABLE Salesman
```

Удаление таблицы из базы данных автоматически не удаляет файл таблицы. Если вы хотите одновременно удалить таблицу из базы данных и DBF-файл с диска, включите опцию **DELETE** в команду **REMOVE TABLE**.

Удалить базу данных с диска можно с помощью Project Manager или командой **DELETE**

DATABASE.

Для удаления базы данных с диска всегда используйте один из вышеуказанных методов. Удаление базы данных с помощью диспетчера проектов или по команде **DELETE DATABASE** автоматически обновляет обратные связи в файлах таблиц базы данных. Если вы удалите БД с помощью какой-нибудь утилиты работы с файлами, то в таблицах, принадлежащих БД, останутся ссылки на уже несуществующую БД, и вы не сможете их использовать.

Команда **DELETE DATABASE** не удаляет файлы таблиц с диска, а только помечает их как свободные. Если вы хотите одновременно с базой данных удалить и файлы таблиц, включите опцию **DELETE TABLES** в команду **DELETE DATABASE**.

В приложении Visual FoxPro можно использовать несколько баз данных одновременно. Для этого либо одновременно откройте все требуемые вам БД, либо сошлитесь на таблицы в закрытых базах данных, как это было показано ранее. Если одновременно открыто несколько БД, вы можете установить текущую базу данных и выбирать таблицы из нее без указания конкретной ссылки.

Как открыть несколько баз данных? В Project Manager выделите имя базы данных и нажмите кнопку **Modify** или **Open**, в зависимости от вашей цели. Или последовательно используйте команду **OPEN DATABASE**.

Когда вы открываете новую базу данных, старые не закрываются. Все уже открытые базы данных так и остаются открытыми, а последняя открытая БД становится текущей.

Все таблицы или другие объекты, которые вы создаете, автоматически становятся частью текущей базы данных. Команды и функции, такие как **ADD TABLE** и **DBC()**, манипулирующие с открытыми БД, оперируют с текущей базой данных.

Вы можете объявить текущей любую из открытых БД с помощью раскрывающегося списка на стандартной панели инструментов Visual FoxPro или командой **SET DATABASE**.

В следующем примере открываются три базы данных, первая объявляется текущей, и ее имя отображается на экране с помощью функции **DBC()**:

```
OPEN DATABASE Auto_Store
OPEN DATABASE Connect_Data
OPEN DATABASE Count_Data
SET DATABASE TO Auto_Store
? DBC()
```

В Visual FoxPro одна или несколько баз данных открываются автоматически при выполнении запроса или формы, в которых используются несколько баз данных. Для того чтобы не потерять управление в программе, явно объявите текущую базу данных.

Как выбрать таблицу из текущей базы данных? Выполните команду **USE** со знаком вопроса "?" В диалоговом окне Use выберите нужную таблицу. Если вам нужна таблица, не входящая в открытую базу данных, в диалоговом окне Use выберите опцию **Other**.

Закрыть базу данных можно в Project Manager или командой **CLOSE DATABASE**. В Project Manager выделите имя базы данных и выберите команду **Close**. В следующем примере закрывается база данных **Auto_Store**:

```
SET DATABASE TO Auto_Store
CLOSE DATABASE
```

Оба указанных способа закрывают базу данных автоматически. Еще один способ закрыть базу данных и другие открытые объекты - включить предложение **ALL** в команду **CLOSE**.

Выполнение команды **CLOSE DATABASE** в окне *Command* не приведет к закрытию базы данных, если она была открыта одним из следующих способов:

- В Project Manager в расширенном режиме просмотра содержимого базы данных.
- Формой, запущенной в собственной Data Session.

В этих случаях БД остается открытой до тех пор, пока она не будет закрыта в Project Manager или пока не будет закрыта соответствующая форма.

В Visual FoxPro текущая база данных используется в качестве первичной области видимости для именованных объектов типа таблиц. Когда база данных открыта, любые запрашиваемые объекты типа таблиц, представлений, связей ищутся сначала в текущей базе данных. Если объект не найден, поиск продолжается в пути по умолчанию.

Например, если таблица **Model** ассоциирована с базой данных **Auto_Store**, приведенные ниже команды всегда найдут таблицу **Model** в базе данных.

```
OPEN DATABASE Auto_Store
```

```
ADD TABLE C:\MYPROJECT\MODEL.DBF
USE Model
```

В следующем примере поиск таблицы **Body** производится в текущей базе данных:

```
USE Body
```

Если таблица **Body** не принадлежит текущей базе данных, ее поиск вне базы будет происходить в пути по умолчанию.

Всегда можно указать полный путь к файлу таблицы, чтобы открывать ее вне зависимости от ее принадлежности к какой-либо базе данных, например, если вас не устраивают изменения в местонахождении таблиц. Однако ссылки на таблицу только по ее имени могут значительно повысить производительность приложения, поскольку в **Visual FoxPro** доступ к именам таблиц из базы данных осуществляется гораздо эффективней, чем по полному пути файла.

Когда вы создаете базу данных, **Visual FoxPro** создает и открывает для монопольного использования файл с именем БД и расширением **DBC (DataBase Container)**. В этом файле хранится вся информация о базе данных, включая имена файлов и объектов, ассоциированных с ней. Сами объекты верхнего уровня типа таблиц или полей в файле **DBC** не хранятся. Хранятся только пути к файлам таблиц.

Для того чтобы увидеть структуру базы данных, вы можете пролистать файл базы данных, посмотреть схему, проверить базу данных и даже расширить файл **DBC**.

Схема базы данных - это визуальное представление структур таблиц и установленных отношений между ними. Схема открытой базы данных отображается в окне Конструктора баз данных.

В Конструкторе баз данных панель инструментов **Database** можно использовать для создания новой таблицы, добавления существующей таблицы в базу данных, удаления таблиц из базы данных, изменения структуры таблицы. Здесь же вы можете редактировать хранимые в базе процедуры.

Файл базы данных содержит записи для каждой таблицы, представления, индекса, индексного тега, установленных отношений и связей, ассоциированных с базой данных; а также записи для каждого поля таблицы или представления, имеющего расширенные атрибуты. В файле хранится и запись, содержащая все хранимые процедуры базы данных.

Конструктор баз данных является концептуальным представлением схемы базы данных, однако вам может потребоваться просмотреть содержимое файла базы данных непосредственно. Можно сделать это, выполнив команду **USE** для файла **DBC** неоткрытой базы данных. В следующем примере открывается окно просмотра, куда выводится содержимое файла базы данных **Auto_Store** в табличном виде.

```
CLOSE DATABASE Auto_Store
USE AUTO_STORE.DBC EXCLUSIVE
BROWSE
```

Не используйте команду **BROWSE** для модификации файла базы данных, если вы не знакомы со структурой файла **DBC**. Любая ошибка при изменении файла **DBC** может разрушить базу данных и привести к потере информации.

В каждом файле **DBC** имеется поле примечаний с именем **User**, в котором вы можете хранить собственную информацию о каждой записи базы данных. Вы можете также расширить файл **DBC**, добавив туда поля, чтобы удовлетворить свои собственные потребности разработчика. Для изменения структуры файла **DBC** вы должны открыть его для монопольного использования.

Как добавить поле к файлу **DBC**?

1. Откройте файл **DBC** для монопольного доступа командой **USE**.
2. Выполните команду **MODIFY STRUCTURE**.

В следующем примере открывается Конструктор таблиц, где вы можете добавить поле в файл **AUTO_STORE.DBC**:

```
USE AUTO_STORE.DBC EXCLUSIVE
MODIFY STRUCTURE
```

При добавлении нового поля в файл базы данных начинайте имя этого поля с символов **"U_"** для обозначения поля пользователя. Это поможет вам избежать конфликтов с любыми возможными изменениями структуры файла **DBC**.

Не изменяйте системные поля в файле **DBC**. Любые такие изменения могут нарушить

целостность базы данных.

Вы можете проверить целостность базы данных с помощью команды **VALIDATE DATABASE**. В следующем примере файл базы данных **Auto_Store** проверяется на целостность:

```
OPEN DATABASE Auto_Store EXCLUSIVE
VALIDATE DATABASE
```

Проверка базы данных гарантирует, что строки файла базы данных хранят правильные метаданные о базе.

Использование словаря данных

Базы данных **Visual FoxPro** используют словарь данных, который повышает эффективность разработки и использования базы данных и освобождает вас от необходимости писать собственные программы обеспечения правил на уровне полей и записей или проверки уникальности значений первичных ключей. Словарь данных **Visual FoxPro** позволяет создавать следующие элементы:

- Длинные имена таблиц.
- Комментарии для каждого поля, таблицы и базы данных.
- Длинные имена полей.
- Заголовки полей, отображаемых в окнах просмотра и в виде заголовков столбцов сетки.
- Значения полей по умолчанию.
- Первичные ключи и ключи-кандидаты.
- Правила на уровне полей и правила на уровне записей.
- Триггеры.
- Постоянные отношения между таблицами базы данных.
- Хранимые процедуры.
- Связи с удаленными источниками данных.
- Локальные и удаленные представления.

Некоторые элементы словаря, такие как длинные имена полей, первичные и ключи-кандидаты, значения по умолчанию, правила на уровне полей и записей, триггеры, хранятся в файле **DBC**, но становятся доступны в процессе создания таблицы или представления.

Вы можете создавать хранимые процедуры для обработки информации в базе данных. Хранимая процедура представляет собой код **Visual FoxPro**, записанный в файле **DBC**. Эти процедуры оперируют только с данными конкретной базы данных. Хранимые процедуры загружаются в память во время открытия базы данных и выполняются намного быстрее, чем процедуры из программных файлов.

Для создания, редактирования или удаления хранимой процедуры в **Project Manager** выделите имя базы данных, выберите в списке пункт **Stored Procedures**, затем нажмите одну из кнопок: **New**, **Modify** или **Remove**.

Для этой же цели в Конструкторе баз данных выберите из меню **Database** команды **Edit Stored Procedures**.

В любом случае откроется текстовый редактор **Visual FoxPro**, в котором вы сможете создать или отредактировать хранимую процедуру текущей базы данных.

Хранимые процедуры используются, в основном, в качестве функций пользователя, на которые можно ссылаться в правилах уровня полей или записей. Когда вы сохраняете функцию в базе данных, код этой функции записывается в файл **DBC** и автоматически перемещается с ней, если вы меняете местоположение базы. Использование хранимых процедур повышает переносимость приложений, поскольку вам не нужно больше управлять файлами процедур пользователя отдельно от файла базы данных.

Поддержание ссылочной целостности предусматривает построение набора правил для сохранения установленных отношений между таблицами при вводе или удалении записей. Аппарат поддержания ссылочной целостности позволяет избежать следующих неприятных ситуаций:

- Добавления записей в подчиненную таблицу, если в главной таблице нет записи с соответствующим первичным ключом.
- Изменения значений в главной таблице, если это изменение приведет к появлению не связанных ("одинокных") записей подчиненной таблицы.

- Удаления записей из главной таблицы, если для нее существуют соответствующие записи в подчиненных таблицах.

Вы можете использовать свои собственные триггеры и хранимые процедуры для поддержания целостности ссылок. Однако Построитель ссылочной целостности Visual FoxPro позволяет определить правила поддержания ссылок, таблицы, для которых необходимо поддерживать целостность, и системные события, после совершения которых будет производиться проверка целостности. Построитель ссылочной целостности управляет многоуровневыми цепочками удалений или обновлений, и настоятельно рекомендуется в качестве инструмента для поддержания ссылочной целостности.

Для работы с Построителем ссылочной целостности откройте Конструктор баз данных. В меню **Database** выберите **Referential Integrity**.

Если вы используете Построитель ссылочной целостности, Visual FoxPro сохраняет генерируемый код в качестве триггеров, которые ссылаются на хранимые процедуры. Вы можете просмотреть код этих процедур, открыв с помощью текстового редактора хранимые процедуры для базы данных, как это было описано выше, но будьте очень осторожны при попытке его модернизации. Может, вы лучше не будете его редактировать?

Если вы изменили проект базы данных, например структуры таблиц, или переназначили индексы, используемые для построения отношений между таблицами, перезапустите Построитель ссылочной целостности, прежде чем начнете пользоваться базой данных. Построитель ссылочной целостности настроит хранимые процедуры и табличные триггеры согласно внесенным изменениям. Если не сделать этого, вы скорее всего получите непредсказуемые результаты, поскольку внесенные в проект базы данных изменения никак не отразятся в ее внутренней структуре.

Вы можете установить постоянные отношения между таблицами базы данных. Постоянные отношения - это отношения, информация о которых хранится в файле базы данных. Постоянные отношения дают возможность выполнять следующие действия:

- Автоматически использовать установленные условия объединения в Конструкторах запросов и представлений.
- В Конструкторе баз данных визуальное отображение связанных индексов таблиц.
- В Конструкторе среды окружения данных (Data Environment Designer) отображение установленных по умолчанию связи для форм и отчетов.
- Использовать аппарат поддержки ссылочной целостности.

В отличие от временных отношений, устанавливаемых по команде **SET RELATION**, постоянные отношения не нуждаются в переопределении при каждом открытии таблиц.

В Visual FoxPro для установки постоянных отношений используются индексы. То есть вы создаете постоянные отношения на основе индексов таблиц, а не на основе связи полей, что позволяет связывать таблицы на основе простых и сложных индексных выражений.

Как создать постоянные отношения между таблицами?

В Конструкторе баз данных выберите индекс, который вы хотите связать, и перетащите его на индекс связываемой таблицы, или включите предложение **FOREIGN KEY** в команду **CREATE TABLE** или команду **ALTER TABLE**.

Если после этого вы посмотрите на схему базы данных в Конструкторе БД, то увидите линию, соединяющую две таблицы и обозначающую новую постоянную связь.

Тип индексного тега или ключа определяет тип устанавливаемого постоянного отношения. Вы должны использовать первичный или индексный тег-кандидат или ключ со стороны "один" в отношении "один ко многим"; со стороны "много" вы должны использовать обычный индексный тег или ключ.

Для удаления постоянного отношения между таблицами в Конструкторе баз данных выделите щелчком линию, соединяющую таблицы. Толщина линии увеличится, отмечая выбор отношения между таблицами. После этого нажмите клавишу **Del**.

Каждая база данных Visual FoxPro содержит свойства **Comment** и **Version**. Вы всегда можете узнать содержание этих свойств с помощью функции **DBGETPROP()**.

В следующем примере на экран выводится номер версии базы данных **Auto_Store**:

```
? DBGETPROP('Auto_Store', 'DATABASE', 'VERSION')
```

Возвращаемое значение является номером версии файла **DBC** и доступно только для чтения.

Программно ввести комментарий в базу данных можно с помощью функции **DBSETPROP()**.

Для этого используйте опцию **COMMENT** в функции **DBSETPROP()**.

В следующем примере вводится новый комментарий для базы данных **Auto_Store**:

? DBSETPROP('Auto_Store', 'DATABASE', 'COMMENT',
'Это моя первая БД')

Функции **DBGETPROP()** и **DBSETPROP()** можно использовать для просмотра и изменения свойств других объектов базы данных, о чем мы расскажем позднее.

Создание и модернизация структуры таблиц

Созданная база данных сможет хранить полезную для пользователя информацию только после того, как в нее будут включены соответствующие таблицы. При создании таблиц уточняются типы данных, заголовки полей, возможные значения по умолчанию, триггеры, а также индексы, которые строятся для организации отношений между таблицами.

Таблицу можно конструировать и создавать в интерактивном режиме с помощью Конструктора таблиц, который доступен из Project Manager или из меню *File*; ее также можно создать программно.

Чтобы создавать и модифицировать таблицы из программ, используются команды, приведенные в табл. 6.2.

Таблица 6.2. Команды создания и модификации таблиц

ALTER TABLE
CLOSE TABLES
CREATE TABLE
DELETE FILE
REMOVE TABLE

Для создания таблицы базы данных в Project Manager выберите базу данных, затем заголовок **Tables** и нажмите кнопку **New**, чтобы вызвать Конструктор таблиц. Окно Конструктора таблиц с необходимыми пояснениями показано на рис. 6.2. При открытой базе данных вы можете выполнить команду **CREATE TABLE**.

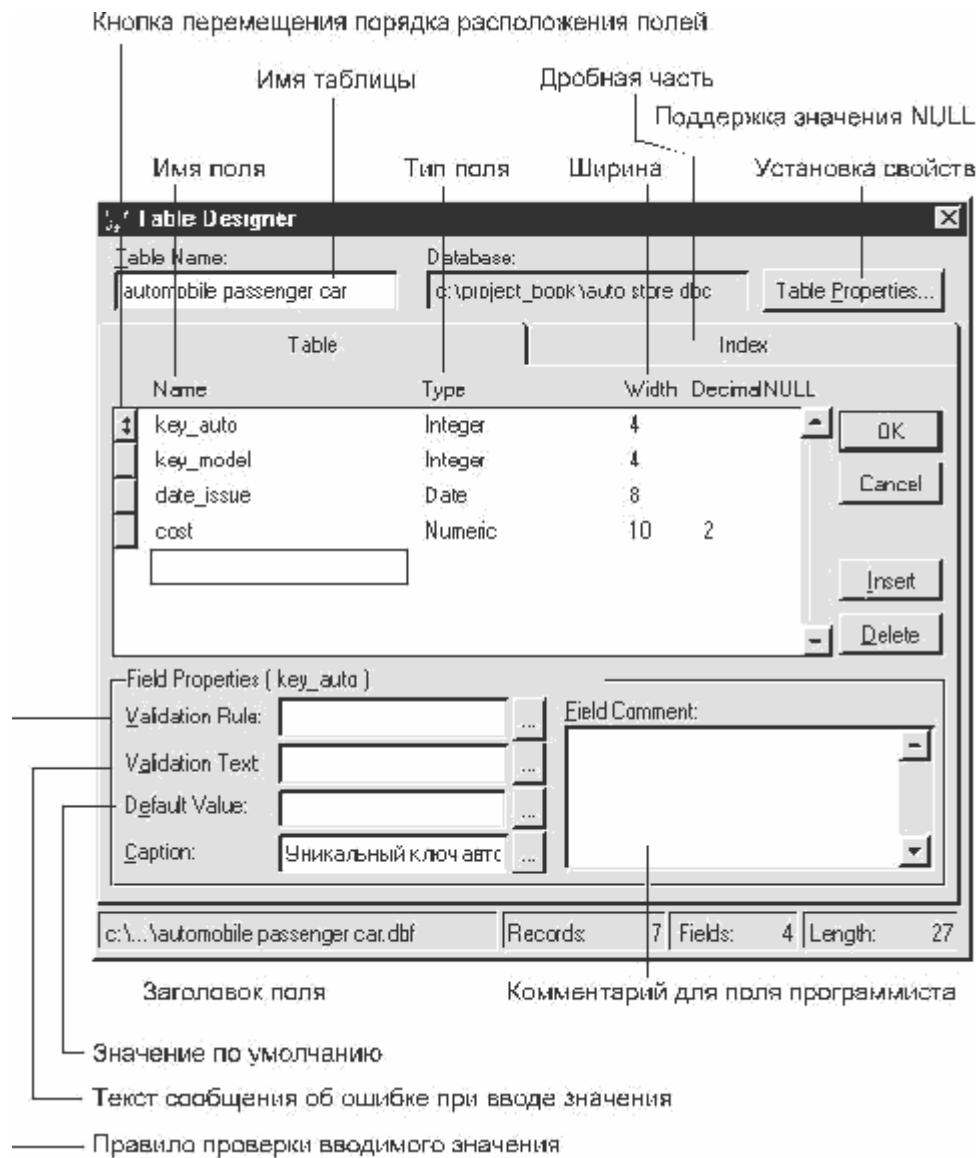


Рис. 6.2.

Следующий фрагмент программы создает таблицу Salesman с четырьмя столбцами под именами key_salman, last_name, first_name, patronymic:

```
OPEN DATABASE Auto_Store
CREATE TABLE Salesman (key_salman I, ;
last_name C(17), ;
first_name C(17), ;
patronymic C(17))
```

Напомним, что в Visual FoxPro символ "I" обозначает поле с типом данных Integer, которое имеет стандартную длину, и она не указывается. Символ "C(17)" обозначает символьное поле с типом данных Character и длиной 17 символов.

Новая таблица автоматически ассоциируется с той базой данных, которая открыта в момент создания таблицы.

Имена таблиц могут состоять из букв, цифр и символов подчеркивания и должны начинаться с буквы или символа подчеркивания.

Если таблица связана с базой данных, можно задать длинное имя таблицы. Длинное имя таблицы может содержать до 128 символов и употребляется вместо стандартного имени для идентификации таблицы в базе данных. Visual FoxPro использует длинные имена таблиц (если они заданы) во всех случаях, когда имя встречается в интерфейсе, например в Конструкторе баз данных, в Конструкторе запросов, Конструкторе представлений, а также в заголовке окна Browse.

Для присвоения таблице базы данных длинного имени в Конструкторе таблиц введите имя в поле Table Name (см. рис 6.2). В программе используйте предложение NAME в команде CREATE

TABLE. Следующий фрагмент создает таблицу **Automobil** и присваивает ей более осмысленное имя **Automobile_Passenger_Car**:

```
CREATE TABLE Automobil NAME Automobile_Passenger_Car ; (key_auto I, ; key_model I, date_issue D, cost N(10,2))
```

Для переименования таблиц или для добавления длинных имен таблицам, которые создавались без них, также используется Конструктор таблиц. Например, если бывшая свободная таблица связывается с базой данных, то с помощью Конструктора таблиц можно присвоить ей длинное имя. Длинные имена могут состоять из букв, цифр, символов подчеркивания и должны начинаться с буквы или символа подчеркивания. В длинных именах не разрешается употреблять пробелы.

В команде **CREATE TABLE** задается имя того DBF-файла, который Visual FoxPro создает для хранения новой таблицы. Это имя файла является стандартным именем таблицы как для таблиц, связанных с базой данных, так и для свободных таблиц.

При создании полей таблицы задается имя поля, тип данных и длина поля. Можно также установить, разрешаются или нет значения **NULL** для поля и значение поля по умолчанию.

Для свободных таблиц имена полей могут быть длиной до 10 символов; для таблиц, связанных с базой данных, имена полей могут содержать до 128 символов. Если таблица исключается из базы данных, то длинные имена полей сокращаются до 10 символов.

Когда создается таблица базы данных, Visual FoxPro запоминает длинные имена полей в записи файла **DBC**. Первые 10 символов длинного имени запоминаются и в файле **DBF** в качестве имени поля.

Если окажется, что усечения длинных имен до 10 символов не являются уникальными в таблице, Visual FoxPro генерирует имена, состоящие из первых *n* символов длинных имен с добавленными к ним последовательными номерами так, чтобы общая длина сгенерированных имен составляла 10 символов. Например, следующие длинные имена будут преобразованы в такие 10-символьные:

Длинное имя	Короткое имя
automobile_passenger_car_key_auto	automobile
automobile_passenger_car_key_model	automobil1
automobile_passenger_car_date_issue	automobil2
...	...
automobile_passenger_car_cost	automobi11

Если таблица связана с базой данных, то ее поля можно обозначать только длинными именами. Использование 10-символьных имен полей для таблицы в составе базы данных невозможно. Если таблица удаляется из базы данных, то длинные имена ее полей теряются и для ссылок на поля таблицы придется использовать 10-символьные имена, хранящиеся в файле **DBF**.

Можно использовать длинные имена полей в индексных файлах. Однако если индекс был создан с использованием длинных имен полей, а затем таблица, для которой он создавался, исключается из базы данных, индекс перестанет работать. В этом случае можно удалить индекс и создать его заново, употребляя короткие имена полей.

Правила составления длинных имен полей такие же, как и для любых идентификаторов Visual FoxPro, за исключением ограничения длины до 128 символов.

После создания в открытой базе данных таблицы можно добавить к каждому ее полю описание, чтобы облегчить понимание и упростить дальнейшую ее модификацию. Visual FoxPro показывает текст комментария при выборе поля в списке полей таблицы в Project Manager.

Для добавления комментария к полю таблицы базы данных в Конструкторе таблиц введите текст комментария в поле ввода **Field Comment** или используйте функцию **DBSETPROP()**.

Например, можно пояснить, что содержится в поле **key_model** таблицы **Model**, введя в качестве комментария к полю текст "Уникальный ключ модели автомобиля":

```
?DBSETPROP('Model.key_model', 'FIELD', 'COMMENT', ;
'Уникальный ключ модели автомобиля')
```

Для каждого поля таблицы в базе данных может быть задан заголовок. Visual FoxPro выводит текст этого заголовка в качестве названия столбца для данного поля в окне **Browse** и в качестве заголовка по умолчанию для колонок объекта **Grid** формы.

Для добавления заголовка поля таблицы базы данных в Конструкторе таблиц введите текст в поле **Caption** в разделе **Field Properties** или используйте функцию **DBSETPROP()**.

Например, можно ввести текст "Наименование модели" как заголовок поля **name_model** в таблице **Model** :

```
?DBSETPROP('Model.name_model', 'FIELD', 'CAPTION',; 'Наименование модели')
```

В процессе создания полей таблицы производится выбор типов данных, для хранения которых предназначаются поля. При определении типа данных для поля следует помнить:

- Какого рода значения допускаются для данного поля (например, в числовое поле нельзя будет заносить текст).
- Объем памяти, резервируемый **Visual FoxPro** для хранения значений поля (например, для каждого значения типа **Currency** используется 8 байтов).
- Какого рода операции могут производиться над значениями данного поля. Например, **Visual FoxPro** может находить суммы значений выражений типов **Numeric** или **Currency**, но не сможет провести эту операцию с выражениями типов **Character** или **General**.
- Сможет ли **FoxPro** выполнять индексирование или сортировку по значениям данного поля (невозможно выполнять сортировку или индексирование по полям примечаний или **General**).

Для телефонных номеров, кодов изделий и других чисел, с которыми не предполагается выполнять математические вычисления, следует задавать тип данных **Character**, а не тип **Numeric**. Если вы используете поле для размещения уникального цифрового ключа, используйте тип данных **Integer**.

При построении новой таблицы может быть задано, что одно или несколько ее полей допускают значения **NULL**. Занесение в поле значения **NULL** констатирует тот факт, что информация, которая обычно должна храниться в данном поле или записи, в настоящее время недоступна.

Как управлять использованием значений **NULL** для отдельных полей? В Конструкторе таблиц установите или снимите отметку со столбца **Null** для данного поля. Если столбец **Null** отмечен, то в поле могут вводиться значения **NULL** (см. рис. 6.2). В программе используйте предложения **NULL** или **NOT NULL** в команде **CREATE TABLE**.

Например, следующая команда создает и открывает таблицу, в которой не допускаются значения **NULL** в поле **key_body**, а разрешаются значения **NULL** в поле **name_body**:

```
CREATE TABLE body (key_body I NOT NULL, name_body C(20); NULL)
```

Можно также определять режим использования значений **NULL** с помощью команды **SET NULL**.

Для того чтобы разрешить использование значений **NULL** во всех полях таблицы в Конструкторе таблиц, отметьте столбец **Null** для каждого поля таблицы или перед использованием команды **CREATE TABLE** выполните команду **SET NULL ON**.

После команды **SET NULL ON** **Visual FoxPro** автоматически отмечает столбец **NULL** для всех полей, создаваемых в Конструкторе таблиц. Если команда **SET NULL** задана до использования команды **CREATE TABLE**, то нет необходимости указывать предложения **NULL** или **NOT NULL**. Например, следующий программный код создает таблицу, в которой допускаются значения **NULL** для всех полей:

```
SET NULL ON
CREATE TABLE Fuel_oil (key_fuel_oil I, name_fuel_oil C(20))
```

Наличие значений **NULL** оказывает влияние на свойства таблиц и индексов. Например, если команда **APPEND FROM** используется для копирования записей из таблицы, где есть значения **NULL**, в таблицу, где они не допускаются, то значения **NULL** будут при копировании рассматриваться или как пустые строки, или как нули, или как пустые поля.

Если требуется, чтобы при добавлении новой записи автоматически заполнялось некоторое поле, то для такого поля можно задать значение по умолчанию.

Значение поля по умолчанию - это число или строка, которые служат стандартным заполнением поля при добавлении новых записей в таблицу, связанную с базой данных.

Значения по умолчанию применяются как при вводе данных с помощью формы, так и при вводе в окно просмотра с помощью представлений или программно и сохраняются в поле, пока не будут введены другие значения.

Значения по умолчанию могут быть заданы в Конструкторе таблиц или программно. Значения по умолчанию могут быть заданы для полей любого типа, кроме General.

Для того чтобы задать значение по умолчанию для поля таблицы, включенной в базу данных, в Конструкторе таблиц введите значение в поле **Default Value**, находящееся в группе **Field Properties** (рис. 6.3), или используйте выражение **DEFAULT** в команде **CREATE TABLE**.

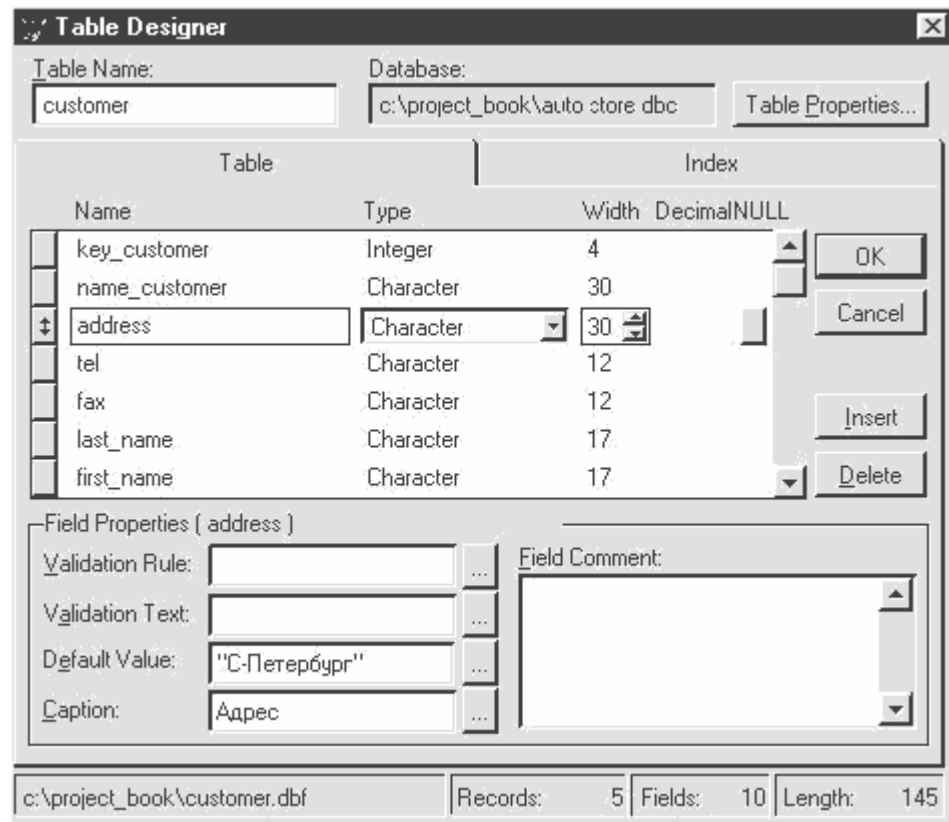


Рис. 6.3. Задание значения по умолчанию в Конструкторе таблиц

Предположим, в силу географического положения магазина чаще всего адрес клиентов данного магазина начинается с фразы "Санкт-Петербург". В следующем примере создается поле **address** со значением по умолчанию "Санкт-Петербург":

```
CREATE TABLE Customer (key_customer I, name_customer C(30), ;
address C(30) DEFAULT "Санкт-Петербург", ;
tel C(12), fax C(12), last_name C(17), first_name C(17),;
patronymic C(17), juridical L, comment M)
```

Если таблица **Customer** уже имеет поле **address**, то добавить для него значение по умолчанию можно следующей командой:

```
ALTER TABLE Customer ALTER COLUMN address SET DEFAULT ;
"Санкт-Петербург"
```

Значения по умолчанию могут использоваться для ускорения ввода данных, позволяя пропускать при вводе поля, для которых не требуется значение, отличное от стандартного.

Если по бизнес-правилам в приложении требуется обязательное наличие значения некоторого поля, то задание значения по умолчанию помогает избежать нарушения соответствующих правил проверки полей и записей.

При удалении таблицы из базы данных удаляются и все связанные с таблицей значения по умолчанию. Хранимые процедуры, на которые ссылались удаляемые значения по умолчанию, сохраняются и после удаления значений по умолчанию.

В качестве значений по умолчанию можно задавать как явные значения в виде чисел, строк символов, дат и т. д., так и выражения, дающие явные значения в результате вычислений. В

качестве такого выражения можно использовать и пользовательскую функцию. В этом случае такую функцию целесообразно записать в хранимых процедурах.

Visual FoxPro производит вычисления для определения типов данных при закрытии определения таблицы. Если тип вычисленного значения несовместим с типом соответствующего поля, то генерируется сообщение об ошибке. Если выражение представляет собой определенную пользователем функцию или содержит вызовы таких функций, то такое выражение не вычисляется.

Для того чтобы обеспечить соблюдение бизнес-правил при вводе данных, можно задать правила проверки полей и записей, которые контролируют допустимость вводимых данных в таблицу базы данных.

Правила проверки полей и записей контролируют вводимые значения на выполнение заданного критерия. Если вводимое значение не удовлетворяет критерию, оно отвергается.

Правила проверки могут задаваться только для таблиц, связанных с базой данных.

Правила проверки полей и записей позволяют контролировать ввод информации как в окне просмотра, так и в экранной форме или при вводе данных из программы. При прочих равных условиях правила проверки позволяют задать контролирующее выражение лаконичнее, чем при использовании выражения **VALID** при проектировании экранной формы или соответствующего кода в программе. Вдобавок, будучи установленными для таблицы базы данных, правила проверки распространяются на все приложения, использующие данную таблицу.

Для контроля вводимых данных можно также определить первичные индексы и индексы-кандидаты, которые предотвращают ввод дублирующих значений полей, а также можно определить триггеры, обеспечивающие целостность ссылок или выполняющие какие-либо действия при изменении данных в базе данных.

Ограничения на уровне поля применяются в тех случаях, когда требуется проверять вводимую в поле информацию независимо от остальной части записи. Например, можно использовать правило проверки поля, чтобы исключить возможность ввода отрицательного числа в поле, которое должно иметь положительное значение. Можно использовать правила проверки поля и для сравнения вводимого в поле значения со значением из другой таблицы.

Не следует задавать такие правила проверки полей и записей, если эти поля и записи являются специфическими для данного приложения. Правила проверки записей и полей должны обеспечивать целостность данных и бизнес-правила, которые всегда применимы к базе данных независимо от приложения. Например, можно задать правило, которое будет сравнивать значение, вводимое в поле **key_body**, со значениями из справочной таблицы кузовов, и отвергать те значения, которые не обнаружатся среди допустимых кодов кузовов.

Как задать правило проверки поля? В Конструкторе таблиц введите выражение критерия в поле ввода **Validation Rule** в группе **Field Properties** (рис. 6.4) или используйте выражение **CHECK** в команде **CREATE TABLE** или **ALTER TABLE**.

Table Designer

Table Name: Database:

Table		Index		
Name	Type	Width	Decimal	NULL
<input type="checkbox"/> key_model	Integer	4		
<input type="checkbox"/> name_model	Character	20		
<input type="checkbox"/> key_firm	Integer	4		
<input type="checkbox"/> swept_volume	Numeric	5	0	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> quantity_drum	Numeric	2	0	<input checked="" type="checkbox"/>
<input type="checkbox"/> capacity	Numeric	5	1	<input checked="" type="checkbox"/>
<input type="checkbox"/> torque	Numeric	5	1	<input checked="" type="checkbox"/>

Field Properties (quantity_drum)

Validation Rule: ...

Validation Text: ...

Default Value: ...

Caption: ...

Field Comment:

c:\project_book\model.dbf | Records: 37 | Fields: 20 | Length: 95

Рис. 6.4. Задание правил проверки и сообщения об ошибке в Конструкторе таблиц

Следующий программный код вводит для таблицы Model правило проверки, требующее, чтобы значения, вводимые в поле `quantity_drum`, были не меньше единицы:

```
ALTER TABLE Model ;
ALTER COLUMN quantity_drum SET CHECK quantity_drum >= 1
```

Если пользователь попытается ввести меньшее значение, Visual FoxPro выведет диалоговое окно, содержащее сообщение об ошибке, и значение будет отвергнуто.

Сообщение, которое выводится при нарушении правил, может быть задано при определении поля. Заданное сообщение будет выводиться вместо стандартного сообщения об ошибке в соответствующем диалоговом окне.

Для задания сообщения о нарушении правил проверки полей в Конструкторе таблиц введите сообщение в поле ввода **Validation Text** в группе **Field Properties** (рис. 6.4) или используйте необязательный параметр **ERROR** в выражении **CHECK** в команде **CREATE TABLE** или **ALTER TABLE**.

Следующий фрагмент программы вводит для таблицы Model правило проверки, требующее, чтобы значения, вводимые в поле `quantity_drum`, были не меньше единицы, а также определяет сообщение об ошибке:

```
ALTER TABLE Model ;
ALTER COLUMN quantity_drum SET CHECK quantity_drum >= 1 ;
ERROR "Количество цилиндров должно быть больше или равно 1"
```

Если пользователь попытается ввести значение меньше 1, Visual FoxPro выведет диалоговое окно с заданным сообщением об ошибке и неверное значение будет отвергнуто (рис. 6.5). Чтобы задать собственное сообщение об ошибке, можно использовать и выражение **SET CHECK** в команде **ALTER TABLE** с необязательным параметром **ERROR**.

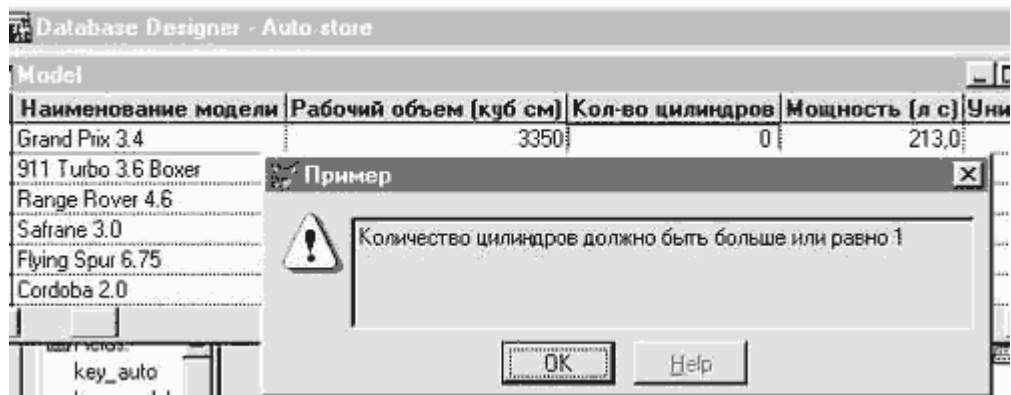


Рис. 6.5.

Не следует включать в правила проверки никаких команд или функций, которые пытаются переместить указатель записи в текущей рабочей области, то есть в той рабочей области, для которой эти правила проверяются. Включение в правила проверки таких команд и функций, как **SEEK**, **LOCATE**, **SKIP**, **APPEND**, **APPEND BLANK**, **INSERT**, **AVERAGE**, **COUNT**, **BROWSE** или **REPLACE FOR**, может привести к повторному срабатыванию правил, создавая ошибочную ситуацию. Такая ситуация называется рекурсией.

В Visual FoxPro триггеры определяются и хранятся как свойства для заданной таблицы. Если таблица удаляется из базы данных, то связанные с ней триггеры тоже удаляются. Триггеры срабатывают после проверки всех прочих ограничений, таких как правила проверки, уникальность первичного ключа, допустимость пустых значений. В отличие от правил уровней полей и записей, триггеры не срабатывают для буферизованных данных.

Триггеры можно создавать в Конструкторе таблиц или с помощью команды **CREATE TRIGGER**. Для таблицы может быть задано по одному триггеру на каждое из событий: **INSERT**, **UPDATE** или **DELETE**. В каждый момент времени таблица может иметь максимум три триггера. Значением триггера может быть истина (.T.) или ложь (.F.).

Для установки триггера в диалоговом окне Table Properties Конструктора таблиц введите в одно из полей ввода - **INSERT**, **UPDATE** или **DELETE Trigger** - выражение триггера или имя хранимой процедуры, содержащей выражение триггера. Программно можно использовать предложение **CHECK** в команде **CREATE TRIGGER**.

Предположим, при продаже очередного автомобиля 27 числа каждого месяца требуется формировать запрос о результатах продажи за текущий месяц. Чтобы этого добиться, можно задать триггер на операцию Insert для таблицы Sale. Триггер устанавливается на операцию **INSERT** (а не на операцию **DELETE** или **UPDATE**), поскольку требуется, чтобы триггер срабатывал при добавлении новой записи. При создании триггера можно указать функцию `__ri_insert_sale()` как триггер вставки таблицы Sale (рис. 6.6). Эту функцию необходимо описать в хранимой процедуре:

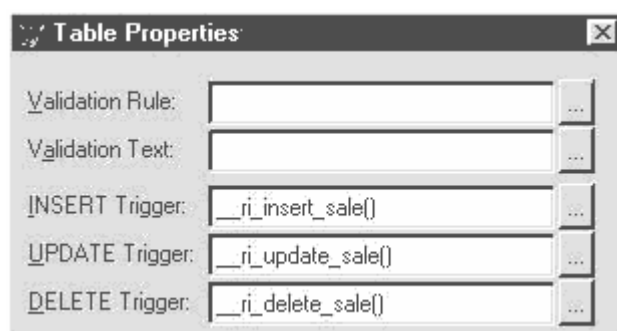


Рис. 6.6.

```

PROCEDURE __ri_insert_sale
    IF DAY(DATE())=27
        WAIT WINDOW 'Формирование отчета' NOWAIT
        SELECT * ;
        FROM Sale ;
        WHERE MONTH(Sale.date_sale) = MONTH(DATE());
        INTO CURSOR Sale_Month_Current
    ENDIF
ENDPROC

```

Триггеры для событий **UPDATE** и **DELETE** задаются аналогично.

После создания таблицы ее структура и свойства могут быть в любое время модифицированы. Для полей можно добавить, изменить или удалить имена, размеры, типы данных, значения по умолчанию, правила проверки, комментарии и заголовки.

Структура таблицы может быть изменена как в среде Конструктора таблиц, так и программно с помощью команды **ALTER TABLE**.

Например, можно изменить структуру связанной с базой данных таблицы **Sale** с помощью следующих команд:

```
OPEN DATABASE Auto_Store
USE Sale EXCLUSIVE
MODIFY STRUCTURE
```

После этого открывается Конструктор таблиц.

Для создания свободной таблицы в **Project Manager** выберите пункт **Free Tables**, а затем нажмите кнопку **New**, чтобы вызвать Конструктор таблиц. Для программного создания свободной таблицы используйте команду **CREATE TABLE** без открытых баз данных.

Следующий программный код создает свободную таблицу **Order_**, состоящую из четырех полей с именами **key_order**, **key_customer**, **key_model**, **key_salman**:

```
CLOSE DATABASES
CREATE TABLE Order_ (key_order I, ;
key_customer I, ;
key_model I, ;
key_salman I)
```

Новая таблица является свободной, поскольку в момент ее создания не было открытых баз данных.

Если требуется перемещаться по записям таблицы, просматривать их или манипулировать ими в некотором заданном порядке, то следует использовать индекс.

Индекс **Visual FoxPro** представляет собой файл указателей, логически упорядоченных в соответствии со значениями индексного ключа. Индексный файл отделен от файла таблицы (**DBF**-файла) и не изменяет физический порядок расположения записей в таблице. При создании индекса создается файл, поддерживающий указатели на записи в **DBF**-файле. Если требуется работать с записями в некотором порядке, то следует выбрать соответствующий индекс, чтобы установить тот порядок, в котором будут видимы и доступны записи таблицы.

Первоначально при создании таблицы **Visual FoxPro** создает файл **DBF** и если имеются поля примечаний и **General**, то связанный с ним **FPT**-файл. В это время не создается никаких индексных файлов. Записи запоминаются в новой таблице в порядке ввода. При просмотре новой таблицы записи видны в том порядке, в котором они вводились. Обычно требуется просматривать записи и иметь к ним доступ в некотором заданном порядке. Например, может потребоваться, чтобы записи в таблице клиентов были отсортированы в алфавитном порядке названий компаний. Если нужно задать порядок доступа к записям и порядок их просмотра, следует создать для таблицы индексный файл путем создания для таблиц первого способа упорядочения или индексного ключа. Тогда можно будет устанавливать порядок записей в таблице в соответствии с индексным ключом и получать доступ к записям таблицы в новом порядке.

Как задать для таблицы индексный ключ? В Конструкторе таблиц на вкладке **Index** введите информацию об индексном ключе. Установите **Regular** в качестве типа индекса или используйте команду **INDEX**.

Например, следующий программный код открывает таблицу **Model** и создает индексный ключ для поля **key_body**. Ключевое слово **TAG** и следующее за ним **"key_body"** задают имя или тег для нового индексного ключа.

```
USE Model
INDEX ON key_body TAG key_body
```

В этом примере тег индексного ключа имел то же имя, что и индексируемое поле. Эти имена не обязательно должны совпадать, для индексного ключа можно задать и другое имя.

После создания с помощью команды **INDEX** индекса **Visual FoxPro** автоматически использует этот индекс, чтобы установить порядок записей в таблице. Например, если ввести данные в таблицу из предыдущего примера и просмотреть ее в окне просмотра, то записи будут отсортированы по полю **key_body**.

При создании первого индексного ключа таблицы в предыдущем примере **Visual FoxPro** автоматически создал новый файл - **MODEL.CDX**, в котором запоминаются индексные ключи.

Индексный CDX-файл, называемый структурным составным индексом, представляет собой наиболее общий и важный тип индексных файлов Visual FoxPro.

Обратите внимание на следующие свойства структурного CDX-файла:

- Автоматически открывается при открытии таблицы.
- В одном индексном файле может содержаться несколько вариантов упорядочения записей или индексных ключей.
- Автоматически поддерживает операции добавления, изменения или удаления записей таблицы.

Visual FoxPro предлагает дополнительно еще два вида индексных файлов: неструктурные CDX-файлы и файлы с одним ключом - IDX-файлы. Однако CDX-файлы, или структурные составные компактные индексы, представляют наиболее важный вид индексов. Остальные два вида индексов употребляются реже.

Visual FoxPro поддерживает четыре типа индексов: первичные, кандидаты, уникальные и обычные. Тип индекса определяет, допустимы или нет повторяющиеся значения полей и записей.

Первичный индекс (Primary) - это единственный индекс в таблице, который определяет уникальное ключевое значение для ее записей.

Следовательно, он запрещает повторяющиеся значения в заданных полях или в выражении. Для таблицы может быть создан только один первичный индекс. При попытке задать первичный индекс для поля, в котором уже имеются повторяющиеся значения, Visual FoxPro выдает сообщение об ошибке.

Индекс-кандидат (Candidate) - это индекс, который удовлетворяет требованиям к первичному индексу, но может быть не единственным в таблице.

Он также запрещает повторяющиеся значения в заданных полях или выражении. Название "кандидат" относится к основному свойству индекса: поскольку он запрещает дублирующие значения, то этот индекс можно использовать в качестве первичного индекса для таблицы.

При попытке задать индекс-кандидат для поля, в котором уже имеются повторяющиеся значения, Visual FoxPro выдает сообщение об ошибке.

Первичные индексы и индексы-кандидаты создаются с помощью команд **CREATE TABLE** или **ALTER TABLE**. При определении постоянного отношения типа "один ко многим" или "один к одному" можно использовать со стороны "один" как первичный индекс, так и индекс-кандидат.

Для создания первичного индекса или индекса-кандидата на вкладке Index Конструктора таблиц выберите тип индекса Primary или Candidate и создайте индекс (рис. 6.7) или используйте команду **ALTER TABLE**.

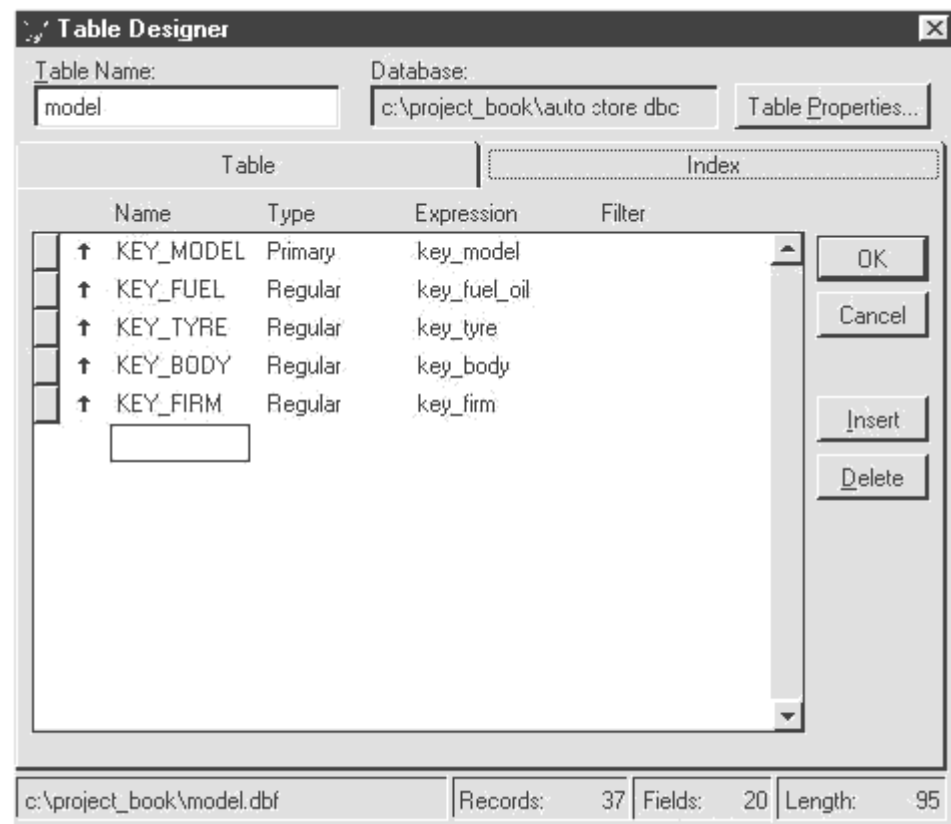


Рис. 6.7.

Например, каждая из следующих команд делает key_model первичным ключом для таблицы Model:

```
ALTER TABLE Model ADD PRIMARY KEY key_model TAG key_mode
ALTER TABLE Model ALTER COLUMN key_model I PRIMARY KEY
```

Первичные индексы и индексы-кандидаты хранятся в структурном CDX-файле данной таблицы, а также в базе данных, когда установлены свойства "Primary" или "Candidate". Эти типы индексов невозможно хранить ни в неструктурных CDX-файлах, ни в IDX-файлах. Основной причиной этого является то, что файл, содержащий индексы таких типов, должен быть всегда открытым, если открыта соответствующая таблица.

Если в индексном выражении, связанном с базой данных, используется определенная пользователем функция, то выражение обрабатывается таким же образом, как и выражения правил проверки и триггеров, содержащие определенные пользователем функции.

В Visual FoxPro уникальный индекс (Unique) не запрещает повторяющиеся значения, однако уникальный индекс запоминает в индексном файле только первое появление каждого значения.

Слово "уникальный" относится к значениям индексного файла, содержащего только уникальные значения ключей; поскольку каждое значение не запоминается более одного раза, все последующие появления какого-либо значения игнорируются. Если таблица проиндексирована уникальным индексом, то она может иметь повторяющиеся значения. Уникальный тип индексов поддерживается для обеспечения совместимости с предыдущими версиями.

Обычный индекс (Regular) - это индекс, который не является уникальным, первичным или индексом-кандидатом.

Обычные индексы используются для упорядочения и поиска записей, но не для обеспечения уникальности данных в этих записях. Обычный индекс используется также при организации

постоянных отношений типа "один ко многим" со стороны многих.

После создания индексных ключей для таблицы **Model** по полям **key_model**, **key_firm**, **key_fuel_oil**, **key_tyre** и **key_body** можно выводить записи в различном порядке, просто выбирая нужный индексный ключ. Команда **SET ORDER** используется для указания того индексного ключа, по которому будет упорядочиваться таблица.

Например, следующие команды открывают окно **Browse** для просмотра данных и выводят записи таблицы **Model**, упорядоченные по полю **key_model**:

```
SET ORDER TO key_model
```

```
BROWSE
```

В табл. 6.3 систематизируются свойства трех видов индексов.

Таблица 6.3. Виды индексов Visual FoxPro

Виды индекса	Описание	Количество ключей	Ограничения
Структурный CDX	Имя файла, совпадает с именем таблицы; автоматически открывается при открытии таблицы	Выражения со многими ключами, называемыми тегами	Максимум 240 символов на вычисляемое выражение
Неструктурный CDX	Должен открываться явно; имеет имя файла, отличное от имени файла таблицы	Выражения со многими ключами, называемыми тегами	Максимум 240 символов на вычисляемое выражение
Простые IDX	Должен открываться явно; имя файла с расширением .IDX определяется пользователем	Одноключевые выражения	Максимум 100 символов на вычисляемое выражение

Неструктурные CDX-индексы применяются в тех случаях, когда требуется создать несколько индексных тегов для какой-либо специальной цели, однако нежелательна постоянная поддержка этих индексов. Например, если приложение содержит некоторый набор отчетов на основе анализа данных в полях, которые в остальное время не индексируются, то прикладная программа может создать неструктурный CDX-индекс с необходимыми индексными тегами, запустить на выполнение эти отчеты, а затем удалить неструктурные CDX-индексы.

Для создания неструктурного индексного тега используйте команду **INDEX** с параметрами **TAG** и **OF**.

Предложение **OF** в команде **INDEX** используется для указания того, что индекс-ный тег должен быть сохранен в файле, отличном от структурного CDX-файла данной таблицы. Например, следующая команда создает тег с именем **name_firm** для таблицы **Firm** и сохраняет его в неструктурном CDX-файле с именем **NSTRFIRM.CDX**:

```
USE Firm
```

```
INDEX ON name_firm TO TAG name_firm OF NSTRFIRM
```

Простой, или отдельный индексный файл, образуется на основе выражений с одним ключом и хранится в файле с расширением **IDX**. В отличие от CDX-индексов, которые строятся на основе выражений со многими ключами, простые **IDX**-индексы сохраняют информацию только об одном ключевом выражении.

Обычно простые индексы используются в качестве временных индексов, создаваемых или реорганизуемых непосредственно перед использованием. Для каждой таблицы может быть создано сколько угодно **IDX**-файлов. Некоторые программисты предпочитают использовать этот тип индекса при создании многопользовательских приложений, так как он легче поддается восстановлению в случае аварийных сбоев.

Вы можете упростить организацию поиска данных в приложении, если воспользуетесь индексами на основе выражений. Выражения могут быть простыми или сложными - это зависит от задачи.

Простые индексные выражения - это индексы, построенные на основе одного поля или сложения нескольких символьных полей для формирования составного ключа. Например, можно создать индекс для таблицы Customer из базы данных Auto_Store на основе выражения:

`last_name + first_name + patronymic`

Если просматривать таблицу Customer, когда она упорядочена по такому ключу, то можно убедиться, что она оказывается отсортированной по полю last_name, затем по полю first_name и только после этого по полю patronymic.

Вы можете создать индекс с помощью следующей команды:

INDEX ON last_name + first_name + patronymic TAG ; LasFirPat

Можно создавать индексы на основе более сложных индексных выражений. Индексные выражения Visual FoxPro могут содержать функции Visual FoxPro, константы или определенные пользователем функции.

Результат вычисления индексного выражения не должен превышать 100 символов для простых (IDX) индексов и 240 символов для тегов CDX-индексов. В одном индексном теге можно комбинировать данные различных типов, преобразовав компоненты выражения к символьному типу.

Для индексных тегов могут употребляться встроенные функции Visual FoxPro. Например, чтобы преобразовать поле типа дата в символьное выражение, можно использовать функцию **DTOS()**, а для преобразования числового значения в строку символов - функцию **STR()**. Пусть требуется создать индексный тег для таблицы Account, в котором бы комбинировались поля date_write (дата выписки счета) и sum_ (сумма оплаты), тогда можно использовать следующее индексное выражение:

INDEX ON DTOS(date_write) + STR(sum_) TAG DatSum

Вы можете расширить возможности индекса, если будете использовать в индексном выражении хранимые процедуры и определенные пользователем функции.

Если индексный тег строится для таблицы, связанной с базой данных, то предпочтительнее использовать хранимые процедуры, а не определенные пользователем функции. Поскольку определенная пользователем функция хранится в файле, отдельном от базы данных, есть вероятность, что этот файл будет перемещен или удален, а это приведет к неработоспособности индексный тег, ссылающийся на такую функцию. Хранимые же процедуры запоминаются в DBC-файле и всегда могут быть найдены Visual FoxPro.

Другим преимуществом использования хранимых процедур в индексных выражениях является то, что в индексе будет гарантировано использование в точности того кода, который указан. Если же в индексном выражении используется определенная пользователем функция, то при выполнении индексирования будет вызываться любая функция с указанным именем, оказавшаяся в области видимости Visual FoxPro.

Использование хранимых процедур или определенных пользователем функций следует тщательно продумывать, поскольку это увеличивает время создания индекса.

Вы можете просматривать записи по убыванию ключа, создав индекс с убывающими ключами или рассматривая обычный индекс в обратном порядке.

Как создать убывающий индекс? На вкладке Index Конструктора таблиц нажмите кнопку со стрелкой слева от поля Name так, чтобы стрелка на кнопке указала вниз, или используйте предложение **DESCENDING** в команде **INDEX ON**.

При создании структурного составного индексного файла можно использовать оба метода. При создании индексов других видов можно использовать только второй способ. Например, можно создать новый убывающий индекс, упорядочивающий таблицу Account от больших к меньшим значениям поля sum_, и просматривать таблицу в таком порядке при помощи следующего программного кода:

```
USE Account
INDEX ON sum_ TAG sum_ DESCENDING
BROWSE
```

Возможность читать индекс по убыванию ключа позволяет воспользоваться существующим индексом, вместо того чтобы создавать новый. Пусть имеется индекс, упорядочивающий таблицу **Account** по полю **sum_**, созданный с помощью следующего программного кода:

```
USE Account
INDEX ON sum_ TAG sum_
```

По умолчанию порядок будет возрастающий. Просматривать таблицу в порядке убывания ключа можно с помощью следующего программного кода:

```
USE Account
SET ORDER TO sum_ DESCENDING
BROWSE
```

В предыдущих примерах внимание было сосредоточено на доступе к информации по убыванию ключа. Однако обе команды - **SET ORDER** и **INDEX** - могут использоваться с опцией **ASCENDING**. Комбинируя разные варианты употребления этих команд, можно добиться значительной гибкости приложения. Например, при создании индекса можно указать опцию **ASCENDING** или **DESCENDING** для наиболее часто используемого порядка, а если необходимо, в команде **SET ORDER** можно указать противоположный порядок.

Вы можете ограничить множество доступных данных только необходимыми данными, используя фильтрующий индекс. Если создать фильтрующий индекс, то видимы и доступны будут только записи, удовлетворяющие критерию фильтрующего выражения.

Как создать фильтр? На вкладке **Index** Конструктора таблиц введите фильтрующее выражение в поле ввода **Filter** нужного индекса (рис. 6.8) или используйте необязательное предложение **FOR** в команде **INDEX**.

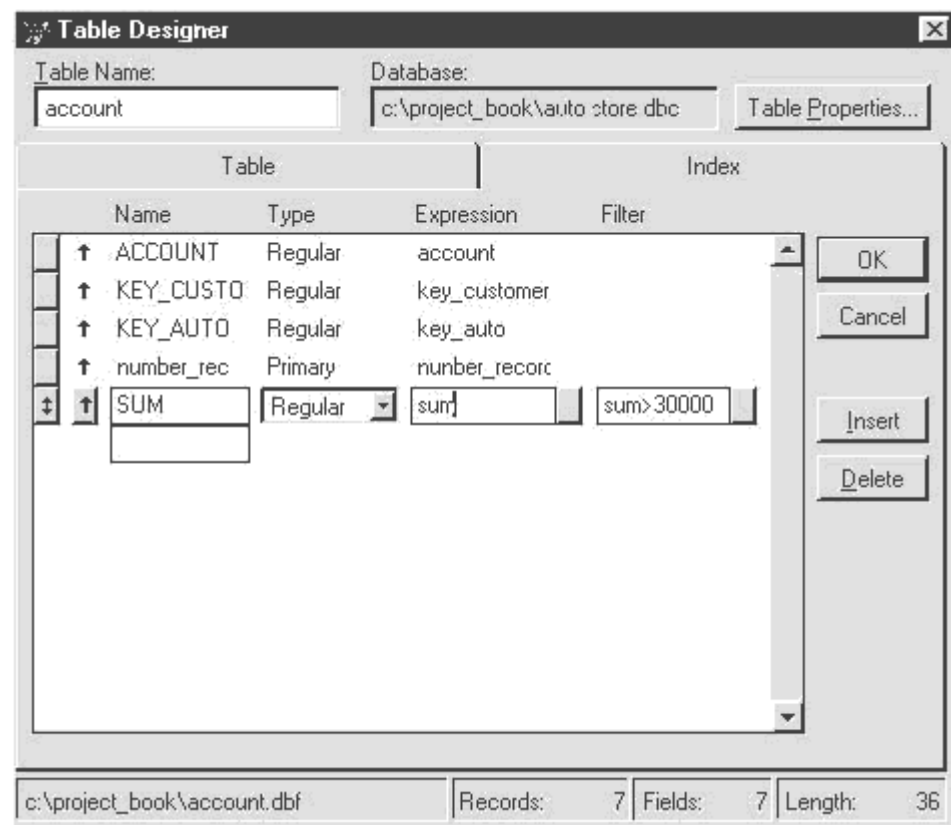


Рис. 6.8.

Если в команду **INDEX** ввести необязательное предложение **FOR**, то индекс будет действовать на таблицу, как фильтр. Индексные ключи будут создаваться в индексном файле только для записей, удовлетворяющих значению фильтра. Следующий программный код создает фильтрующий индекс и выводит отфильтрованные данные в окне просмотра:

```
USE Account
INDEX ON sum_ FOR sum_ > 30000 TAG sum_
```

BROWSE

Для того чтобы временно отфильтровать данные без построения специального фильтрующего индекса, используется команда **SET FILTER**. Особенно полезна эта команда в тех случаях, когда требуется задать временное условие, которому должны удовлетворять записи таблицы, чтобы быть доступными. Например, чтобы отфильтровать таблицу **Account** так, чтобы показать только счета с суммой оплаты больше 30000, можно использовать следующие команды:

```
USE Account
SET FILTER TO sum_ >> 30000
BROWSE
```

Команда **SET FILTER** воспринимает в качестве фильтра любое допустимое логическое выражение **Visual FoxPro**. После того как выдана команда **SET FILTER**, в таблице будут доступны только записи, удовлетворяющие условию фильтра. Все команды доступа к таблицам действуют с учетом установленного командой **SET FILTER** критерия. Для каждой открытой таблицы может быть установлен свой фильтр.

Вы можете повысить производительность при работе с индексируемыми таблицами, поддерживая индексы в актуальном состоянии (соответствующим данным в таблицах) и используя в них оптимизируемые выражения.

Индексный файл может оказаться устаревшим, если таблица открывается без соответствующего индексного файла и в ее ключевые поля вносятся изменения. Индексные файлы могут также оказаться недействительными в результате аппаратных сбоев, а также в результате редактирования таблицы программами, отличными от **Visual FoxPro**. Если индексный файл оказался устаревшим, то его можно обновить командой **REINDEX**.

Как перестроить индексный файл? В меню *Table* выберите *Rebuild Indexes* или используйте команду **REINDEX**.

Например, следующий программный код обновляет индексный файл для таблицы **Account**:

USE Account REINDEX

Команда **REINDEX** обновляет все индексные файлы, открытые в текущей рабочей области. **Visual FoxPro** распознает все виды индексных файлов (составные CDX-файлы, структурные CDX-файлы и простые IDX-файлы) и соответственно их перестраивает. Обновляются все теги в CDX-файлах и структурных CDX-файлах, открывающихся автоматически вместе с таблицей.

Перестройка индексов требует временных затрат, особенно для больших таблиц. Перестройку индексов следует производить только при необходимости.

Итак, пришло время вернуться к нашей задаче, описанной в [первой главе](#) (раздел 1.2) и спроектированной во второй (раздел 2.2). Следующий программный код описывает создание базы данных **Auto_Store**. Полностью программа записана на прилагаемой к книге дискете, здесь мы приводим ключевые фрагменты, исключая однотипные операции:

WAIT WINDOW "Один момент..." NOWAIT

* После выполнения данного кода не забудьте БД

* "Auto_Store" включить в проект (с помощью клавиши Add,
* предварительно выделив пункт Databases во вкладке Data)

CREATE DATABASE auto_store && Создание БД Auto_Store

? DBSETPROP('auto_store', 'database', 'comment', ;

'Автоматизация управления работы дилера по продаже легковых автомобилей.') && Комментарий
к БД

* Создание таблицы Model

CREATE TABLE Model (key_model i, name_model c(20), key_firm i DEFAULT 1, ;

swept_volume n(5) NULL, quantity_drum n(2) NULL, ;

capacity n(5,1) NULL, torque n(5,1) NULL, ;

key_fuel_oil i DEFAULT 1, top_speed n(5,1) NULL, ;

starting n(4,1) NULL, key_tyre i DEFAULT 1, ;

key_body i DEFAULT 1, quantity_door n(1) NULL, ;

quantity_sead n(2) NULL, length n(5)<|>NULL, width n(4) ;

NULL, ;

height n(4) NULL, expense_90 n(4,1) NULL, ;

expense_120 n(4,1) NULL, expense_town n(4,1) NULL)

* Установка заголовков (для таблицы model)

? DBSETPROP ('Model.key_model', 'field', 'caption', ; 'Уникальный ключ модели')

? DBSETPROP ('Model.name_model', 'field', 'caption', ; 'Наименование модели')

```

? DBSETPROP ('Model.key_firm', 'field', 'caption', ;
'Уникальный ключ фирмы')
...
* Создание комментария (для таблицы Model)
? DBSETPROP ('Model.key_firm', 'field', 'comment', ;
'По данному полю можно определить наименование фирмы ; через таблицы Firm')
* Создание первичного ключа (для таблицы Model)
ALTER TABLE Model ADD PRIMARY KEY key_model TAG key_model
* Формирование правил и сообщений об ошибке
* (для таблицы Model)
ALTER TABLE Model ALTER COLUMN swept_volume SET CHECK ; swept_volume > > 0 ;
ERROR "Значение рабочего объема не должно быть ; отрицательным"
ALTER TABLE Model ALTER COLUMN quantity_drum SET CHECK ; quantity_drum > >= 1 ;
ERROR "Количество цилиндров должно быть больше или равно 1"
* Создание таблицы Firm
CREATE TABLE Firm (key_firm i, name_firm c(20), ; key_country i)
* Установка заголовков (для таблицы Firm)
? DBSETPROP ('Firm.key_firm', 'field', 'caption', ;
'Уникальный ключ фирмы')
? DBSETPROP ('Firm.name_firm', 'field', 'caption', ;
'Наименование фирмы')
? DBSETPROP ('Firm.key_country', 'field', 'caption', ;
'Уникальный ключ страны')
* Создание первичного ключа (для таблицы Firm)
ALTER TABLE Firm ADD PRIMARY KEY key_firm TAG key_firm
...
* Создание таблицы Automobile_Passenger_Car
CREATE TABLE Automobile_Passenger_Car (key_auto i, ; key_model i, date_issue d, cost n(10,2))
* Установка заголовков
* (для таблицы Automobile_Passenger_Car)
? DBSETPROP ('Automobile_Passenger_Car.key_auto', ; 'field', 'caption', 'Уникальный ключ
автомобиля')
...
* Создание первичного ключа
* (для таблицы Automobile_Passenger_Car)
ALTER TABLE Automobile_Passenger_Car ADD PRIMARY KEY ; key_auto TAG key_auto
* Создание постоянных отношений
ALTER TABLE Automobile_Passenger_Car ;
ADD FOREIGN KEY key_model TAG key_model REFERENCES Model
ALTER TABLE Model ;
ADD FOREIGN KEY key_firm TAG key_firm REFERENCES Firm
ALTER TABLE Model ;
ADD FOREIGN KEY key_fuel_oil TAG key_fuel REFERENCES ; Fuel_Oil
...

```

6.2. Access

Согласно сложившемуся мнению, СУБД Microsoft Access предназначена для конечных пользователей, которые легко и непринужденно создают достаточно сложные приложения. Строят связи между таблицами, создают изощренные запросы и на их основе проектируют формы и отчеты. Если нужен какой-нибудь "наворот", то используют макросы. Сразу оговоримся, что это вполне возможно, хотя мы уверены, что без программирования даже в Access не обойтись.

В этом параграфе мы рассмотрим визуальные методы создания баз данных в MS Access.

Основой любой системы обработки систематизированных данных являются таблицы. Казалось бы, вслед за этим может последовать вопрос: а почему бы не использовать Excel, в котором таблицы и создавать не надо, они уже давно созданы. Авторам известен случай, когда директор одного из заводов, страшный поклонник Excel, требовал от специалистов по информационному обеспечению построить автоматизированную систему управления документооборотом на основе Microsoft Excel. Критерием приема на работу на этом заводе программиста, да и любого другого специалиста, было наличие твердых и уверенных познаний электронных таблиц известного производителя программных продуктов. К сожалению, до конца проследить за этой историей не удалось, возможно, что-нибудь получилось. Да и почему нет - в конце концов, живем в свободной рыночной стране, если сказали, что это АСУ, значит, это АСУ, и не рассказывайте нам сказки про связи, сущности и уж тем более про ODBC.

В Access можно сконструировать 90 % приложения из данных вам "кубиков", которые здесь

называются объектами, и дописать немножко кода для особо требовательных процессов.

Возможны контраргументы - "больно медленный продукт". Уверяем вас, не медленней, чем Delphi, когда тот работает с данными. Не делает EXE-файлы. Минус, но насколько важный? Время, когда надо было "работать с дисководом", уже прошло.

Мы не будем никого разубеждать и что-либо доказывать. Наша цель - показать, как строить базы данных в Access и как работать с ними.

Access - достаточно тесно интегрированный в Microsoft Office продукт. Попро-буйте провести следующую операцию. Откройте любую таблицу в Excel. Наберите какие-нибудь табличные данные. Например, как в следующей таблице:

First	Second	Third
Вольво	2	12.03.96
Волга	56	12.04.96
Мерседес	34	17.12.96
Тойота	78	14.09.23

Теперь выделите область с данными и скопируйте ее в буфер. Если у вас еще не открыт Access, то откройте его. Для того, чтобы создать контейнер базы данных, выберите в меню *Файл* команду *Создать*. Либо воспользуйтесь значком с изображением белого листка бумаги, во всех приложениях Microsoft означающим "создать новый файл". В контейнере, который представляет собой графическое средство работы с базой данных, отображаются все объекты, которые в ней содержатся.

Объекты каждого типа располагаются на своих страницах. Нас интересуют таблицы, поэтому переходим на страницу Таблицы. Выполнив операцию Вставить, мы получаем новую таблицу. У нее есть один недостаток, на который в принципе не стоит обращать внимания, - текстовые поля всегда имеют длину в 255 символов.

Мы не призываем вас сразу же переводить таблицы из Excel в Access, это был просто маленький пример, показывающий интеграцию Access в Microsoft Office.

Первое, с чего рекомендуется начинать - создание таблицы. Очевидно и, можно сказать, банально. Следует отметить одно "но" - структура вашего приложения может быть достаточно сложной и не всегда вписываться в рамки простых стандартов. Приведем несколько примеров.

Вы используете данные с сервера, в таком случае, возможно, вам не придется создавать таблицы, а надо присоединять таблицы с сервера. Эта операция не намного сложнее, чем их создание.

Подобная же ситуация может сложиться, если на вашем предприятии применяют приложения, использующие другие процессоры данных. Вы можете совместно обрабатывать их, используя присоединенные таблицы. То есть вы работаете на Access, а все остальные, например, на FoxPro. При этом используете одни и те же таблицы в формате FoxPro 2.6.

Рассмотрим другой случай. Вы решили хранить таблицы в одной базе данных, а объекты их обработки и вывода - формы и отчеты - в другой. В таком случае вы либо используете присоединенные таблицы, либо программным способом создаете запросы к таблицам и используете их в своих формах.

Для начала рассмотрим обычный случай, который может охватить достаточно широкий круг задач. В Контейнере БД перейдите на вкладку Таблицы и нажмите на кнопку Создать. Перед вами появится диалоговое окно Новая таблица, показанное на рис. 6.9. В списке справа перечислены пять пунктов, которые предоставляют различные варианты создания таблицы.

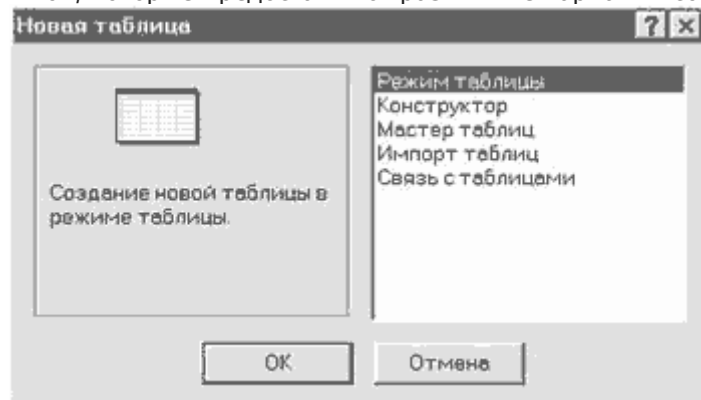


Рис. 6.9.

В первом случае на экран будет выведена готовая таблица для заполнения ее данными, как показано на рис. 6.10.

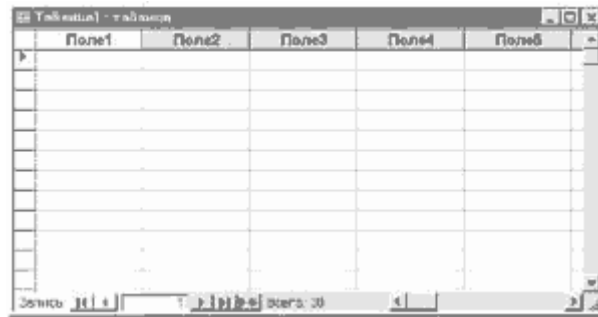


Рис. 6.10.

Во втором случае на экране появляется Конструктор таблиц, окно которого показано на рис. 6.11. Вооруженные знаниями типов полей, которые мы получили в [главе 3](#), можем перейти к конструированию. Для этого в первой колонке нам надо ввести название поля, а во второй, как следует из ее названия, выбрать тип из предлагаемого списка. В третьей колонке настоятельно рекомендуем как можно подробнее описать, с какой целью это поле появилось в таблице. Если таблиц много, то вы можете проявлять сколь угодно бурную фантазию при выборе названия поля и все равно однажды не сможете вспомнить, а для чего какое-то поле появилось. Обратим ваше внимание на полезную информацию, которая выводится в правом нижнем углу. Вы можете узнать, что бесполезно давать полю имя, число символов в котором более 64. А если нажмете на клавишу F1, то узнаете, что нельзя использовать точку, восклицательный знак и квадратные скобки. Пробелы могут быть внутри названия, но нельзя использовать их в начале имени.

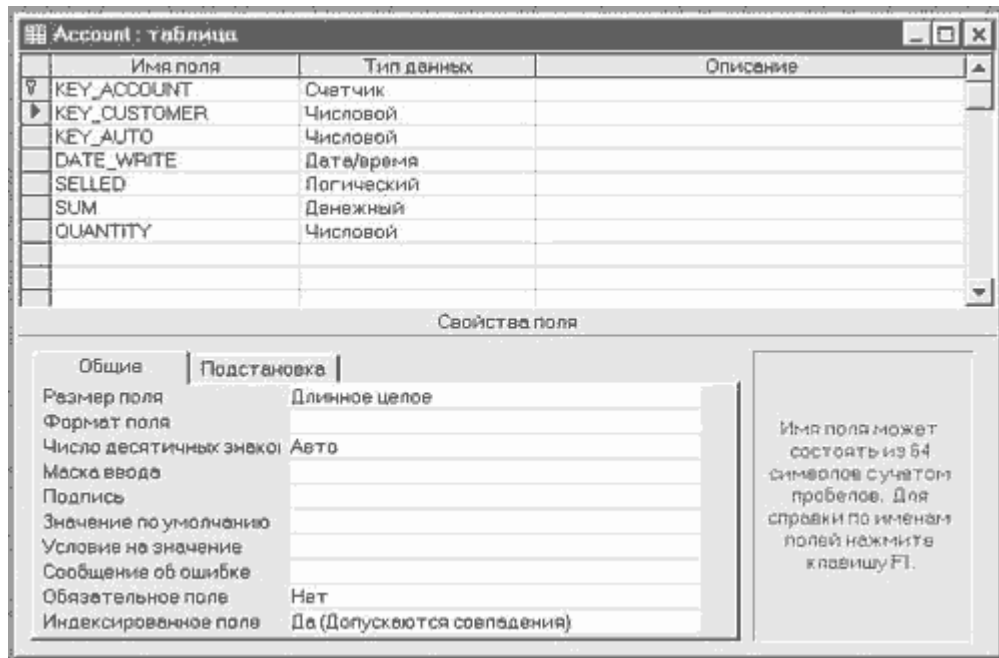


Рис. 6.11.

Для каждого типа поля в левой нижней части экрана высвечивается свой набор свойств. Для текстового поля обязательно укажите длину. Размеры по умолчанию вновь создаваемых текстовых полей можно установить с помощью диалогового окна **Параметры**, вызов которого происходит посредством задания одноименной команды в меню **Сервис**. В этом диалоге выберите вкладку **Таблицы/запросы**, а в ней область с заголовком **Размеры полей по умолчанию**, как это продемонстрировано на рис. 6.12. Опыт подсказывает, что довольно часто найти оптимальную длину поля невозможно, поэтому какое бы значение по умолчанию вы ни установили, свойство "длина поля" редактировать придется часто. Главный принцип, которым вы должны руководствоваться, - самое длинное значение, которое вы введете в это поле, должно "чувствовать себя в нем комфортно" и не быть усеченным. Можете особо не жалеть места. Структура файла MDB такова, что лишнее пространство в поле, не занятое символами, не хранится.

Можете провести эксперимент. Увеличьте размеры текстовых полей, но не редактируйте записи. Потом проверьте размер. Он изменится. Но не навсегда - есть способ вернуть его к прежнему размеру. Необходимо использовать сжатие базы данных. Эта операция доступна с помощью меню **Сервис**, в котором есть команда **Служебные программы**, одна из которых и

сможет сделать базу данных более компактной. Эту операцию надо проводить периодически, так как, даже если вы удалите таблицу размером в пять мегабайт, ваша база данных автоматически не уменьшится, пока вы не выполните сжатие. Меню **Сервис** динамически меняет свое содержимое и для того, чтобы увидеть в нем команду **Служебные программы**, необходимо закрыть базу данных.

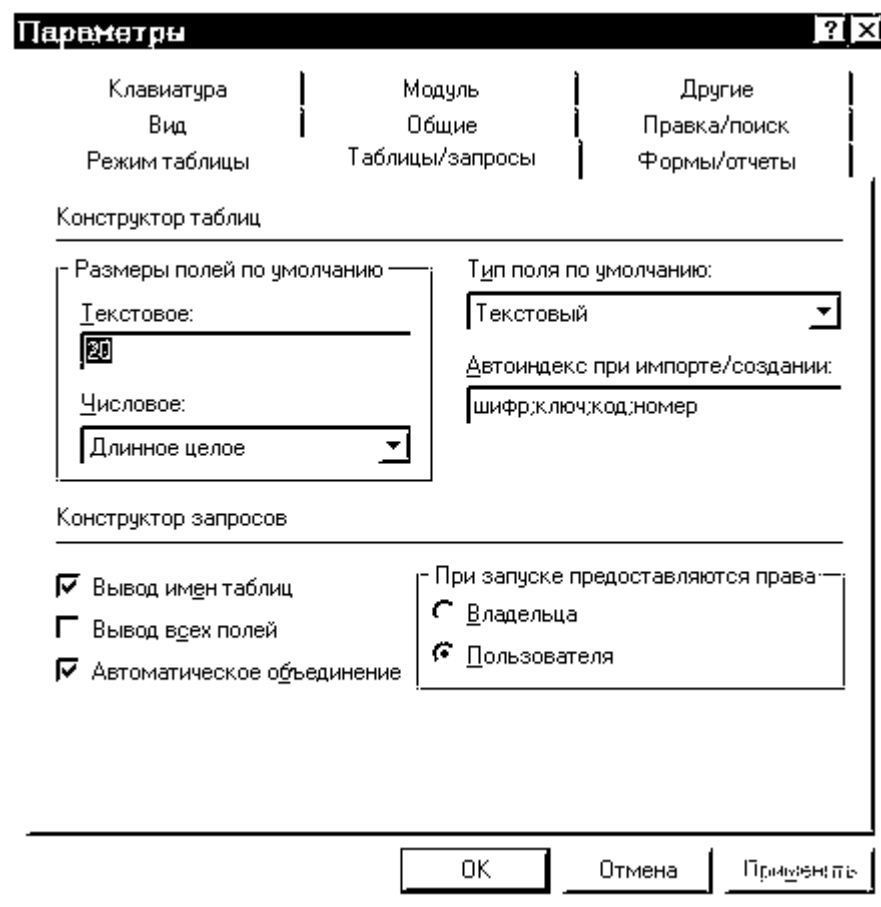


Рис. 6.12. Установка размеров поля в Access по умолчанию

Следующее свойство поля - **Формат** - служит для вывода значений поля в более удобном виде. Например, вам хочется выводить телефонные номера в общепринятом виде, то есть вот так: (812)259-4277, а хранить в таблице в виде того же набора цифр, но без скобок и дефисов. Тогда наберите в строке **Формат** следующее выражение:

(000)00-00000

Теперь данные будут выводиться так, как мы привыкли видеть телефонные номера, хотя храниться будут совсем по-другому. Для данного поля лучше использовать текстовый тип данных. Поэкспериментируйте - поймете почему.

Когда мы начнем вводить данные или редактировать их, то снова увидим строку типа 8125250495. Эту проблему можно решить с помощью свойства **Маска ввода**. Наберите то же самое, что и для свойства **Формат**. Access сам отредактирует введенное вами значение, оно будет выглядеть так:

\(0\-000\)000\-00\-00

Если вы хотите подробнее ознакомиться с символами, которые используются при создании значений свойств **Формат** и **Маска ввода**, то приготовьтесь потратить время - их достаточно много. На первых порах советуем вам распечатать темы файла контекстной справки по этим свойствам.

Для некоторых типов полей вы можете воспользоваться встроенными значениями форматов. Особенно много их у поля типа "дата и время".

Следующее свойство - это **Подпись поля**, в английской версии **Caption**, то есть то, что мы привыкли называть заголовком. Подпись - это альтернатива названия поля. При этом

ограничение на количество символов значительно превосходит аналогичное для название поля - здесь можно написать целое произведение, так как вы можете использовать 2048 символов (к примеру, бесплатное объявление в Санкт-Петербургской газете "Реклама-Шанс" не должно превышать 80 символов). Если вы введете значение для этого свойства, то при просмотре таблицы оно будет выводиться вместо заголовка поля, в противном случае, как вы уже поняли, будет выводиться название поля.

Свойство Значение по умолчанию удобно использовать в различных случаях. Допустим, вы хотите знать дату и время появления новой записи в вашей таблице. В таком случае сделайте значение поля по умолчанию равным функции **Now()**. Теперь вы четко можете отследить, в какой момент была добавлена новая запись.

В сочетании со свойством Условие на значение, свойство Значение по умолчанию становится еще более сильным ограничителем для желающих вводить фальсифицированные данные. Установите свойство Условие на значение равным следующему выражению:

`<=Now() And >>Now()-1`

Теперь никто не сможет отредактировать данные в этом поле и, к примеру, провести какую-нибудь операцию задним числом.

Все это, конечно, не совсем так просто. Но тем не менее вам удастся навести порядок в обработке ваших данных, а в сочетании со средствами защиты вы сможете предохранить свои данные от любых дальнейших изменений, не каждый любитель так называемого "взлома" сможет добраться до них.

После того как пользователь попытается нарушить Условие на значение, выведется стандартное сообщение Microsoft Access. Это можно легко преодолеть, установив значение свойства Сообщение об ошибке. Учтите, что свойство имеет строчный тип данных, поэтому данное значение вводите в кавычках. Можете сделать его равным значению, возвращаемому какой-либо функцией, естественно, оно тоже должно быть строчным.

Свойство Обязательное поле требует, чтобы в поле было введено какое-нибудь значение.

Свойство Пустые строки не противоречит свойству Обязательное поле. Если вы установите значение этого свойства равным "Да", то сможете вводить значения типа NULL, то есть отсутствие какого-либо значения.

Каждое поле, кроме полей примечаний и объектов OLE, может быть индексировано, что значительно ускоряет поиск, но замедляет ввод и обновление данных. Для того чтобы создать индекс, используйте свойство Индексированное поле, при этом вы можете создать индекс, который не может иметь дублирующих друг друга значений в разных записях или, напротив, допустить повторение значений в индексированном поле. Но таким образом вы можете создать только простые индексы, которые состоят из значений одного поля. Для того чтобы создать сложные индексы, используйте команду **Индексы** из меню **Вид**. Здесь вы можете в левой колонке ввести название индекса, а во второй колонке последовательно указать поля, которые добавляете в индекс (рис. 6.13).

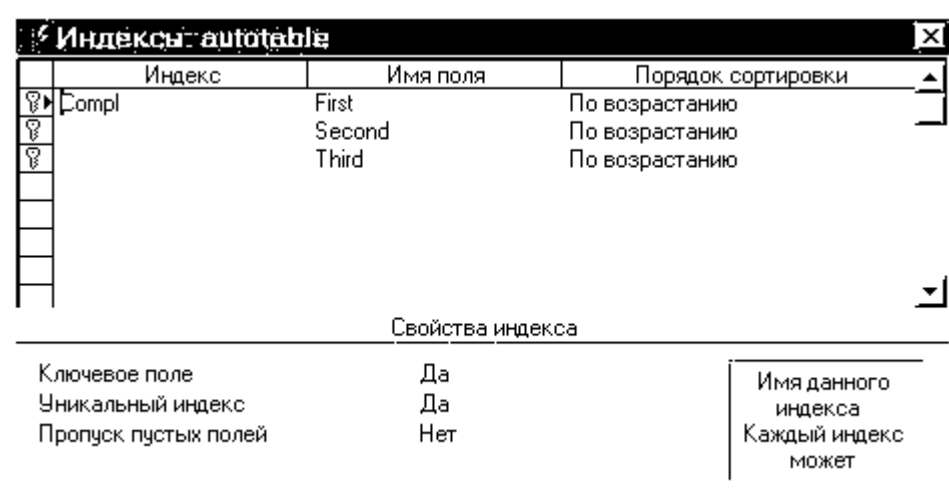


Рис. 6.13. Создание сложного индекса в Access

Можно сделать поле первичным ключом, а затем использовать его для связи таблицы с другими таблицами. Причем это будет связь, которая позволяет использовать контроль целостности данных.

Когда вы будете создавать базу данных, вам придется очень много общаться с заказчиком или постановщиком задачи и чертить различные схемы, обсуждать, где какие данные будут

храниться, как таблицы будут связываться. Если у вас еще не выработался универсальный способ общения с пользователем и воплощения выработанных идей, используйте Схему данных. Для того чтобы вывести ее на экран выберите команду *Схема данных* в меню **Сервис** или значок с изображением связанных таблиц в Стандартной панели инструментов.

Из схемы данных, которая приведена на рис. 6.14, вы легко можете попасть в Конструктор таблиц. Для этого выделите таблицу и нажмите правую клавишу. В появившемся всплывающем меню выберите нужную команду. Для того чтобы связать две таблицы, выберите поле в графическом изображении таблицы и с помощью мыши перетащите на соответствующее поле в таблице, с которой вы связываетесь. Обычно для связи используют ключевые поля, так как в этом случае становятся доступными средства контроля целостности данных, например каскадное обновление и каскадное удаление связанных записей.

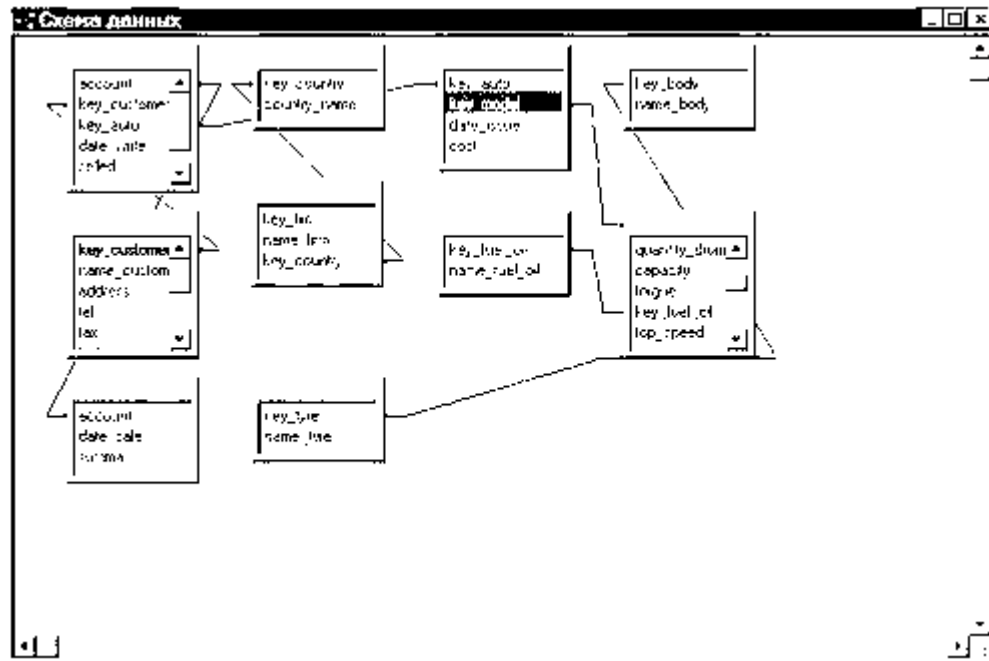


Рис. 6.14. Графическая схема созданной базы данных в Access

Все вышеизложенное можно проделать с помощью объектов доступа к данным, то есть создать таблицы и установить их свойства с помощью кода. Для того чтобы узнать об этом, обратитесь к следующему параграфу данной главы, в котором описывается построение БД в Microsoft Visual Basic.

Создав, подключив или импортировав таблицы, установив правила для полей, построив индексы и ключи и на их основе постоянно хранимые связи, кое-где, к тому же, с контролем целостности данных, вы решили львиную долю задачи. Правда, приходится отметить, что пожелания пользователя могут меняться до самого последнего момента разработки, так что к этому тоже надо быть готовым.

Создав таблицы, вы можете воспользоваться средством Анализ таблиц, который позволяет разбить данные из одной таблицы на несколько, если это необходимо. При этом Access руководствуется принципом избыточности данных, то есть если он заметит, что какие-то значения часто повторяются, то предложит вынести их в отдельную таблицу. Соответственно, для того чтобы таблицы могли разделяться, они должны содержать данные. Этот способ рекомендуется использовать при импорте данных из других приложений и переносе таблиц Excel в базу данных Access.

6.3. Visual Basic

Visual Basic, являясь универсальным средством разработки пользовательских программ, предоставляет весьма скудные средства для визуальной работы с БД. Это не покажется вам неестественным, если вы помните, что он использует общий с Access процессор БД и, следовательно, не имеет значения, создана БД в Access или Visual Basic. Для работы с другими форматами данных можно использовать ODBC.

В этом параграфе мы рассмотрим программные методы создания БД в Visual Basic, обращая особое внимание на такой важный компонент, как объекты для доступа к данным - DAO.

Visual Basic позволяет использовать два метода для работы с данными:

- с помощью специальных элементов управления для работы с данными;
- путем применения объектов для доступа к данным (DAO - Data Access Objects).

Если элементы управления данными позволяют практически без программирования обеспечить элементарные возможности по просмотру и редактированию информации в существующих БД, то использование объектов для доступа к данным хотя и требует программирования, но зато предоставляет самые широкие возможности по управлению базой данных. При этом не стоит думать, что программист перед проектированием прикладной программы должен выбрать один из методов. Наоборот, почти наверняка в своей прикладной программе вы будете использовать как один, так и второй способ работы с данными. В дальнейшем мы постараемся дать четкие рекомендации по использованию этих двух методов в различных ситуациях, возникающих при проектировании прикладной программы. В том случае, если вам интересны технологии клиент-сервер, обратите внимание на специальные методы, которые могут использоваться для доступа к данным в этом случае и которые описаны в [главе 8](#).

Модель управления данными, основанная на объектах DAO, представляет собой коллекцию классов, которые моделируют структуру реляционной базы данных.

Эти объекты имеют свойства и методы, которые позволяют выполнять все необходимые операции для управления базой данных, включая ее создание, описание таблиц, полей и индексов, связей между таблицами, перемещения по записям таблиц, выполнения запросов и т. д. Внутренний механизм процессора данных транслирует эти операции с объектами для доступа к данным в физические действия над файлами БД.

Visual Basic имеет две отдельные версии процессора данных и библиотек объектов для доступа к данным:

- версия 2.5 предназначена для написания программ, работающих в 16-битовой операционной системе (Windows 3.x и Windows for Work-groups);
- версия 3.0 - для 32-битовой ОС (Microsoft Windows 95 или Windows NT).

Написание прикладной программы для обработки данных в Visual Basic схематично состоит из создания объектов для доступа к данным, таких, например, как Database, TableDef, Field и Index, которые соответствуют различным составляющим физической БД, к информации которой вы хотите получить доступ. Используя соответствующие свойства и методы созданных объектов, можно выполнять необходимые операции с базой. Увидеть результат выполнения операций и обеспечить редактирование данных пользователем можно с помощью экранных форм, в которых будут использоваться соответствующие элементы управления.

Изложенный подход упрощает вашу прикладную программу и позволяет обойтись значительно меньшим количеством строк кода, написанного вручную, а так же избавляет от необходимости непосредственного манипулирования и поиска необходимых данных. Программа становится более гибкой, потому что вы можете использовать те же самые объекты, свойства и методы при работе с разнообразными поддерживаемыми форматами баз данных. И если вам необходимо перейти от одного формата базы данных к другому (например, при перенесении базы данных Visual FoxPro на SQL Server), для этого потребуются сделать лишь небольшие изменения в программном коде. Таким образом вы даже можете создавать прикладные программы, которые будут соединять таблицы из двух и более различных баз данных в одном запросе или отчете.

С помощью Visual Basic вы получаете доступ к БД, которые можно разбить на три категории:

- Базы данных Visual Basic, используют формат, аналогичный Microsoft Access. Управление этими БД осуществляет непосредственно процессор данных Visual Basic, что обеспечивает программе максимальную гибкость и скорость работы.
- Доступ к форматам БД внешних СУБД для персональных систем, таких как Btrieve, dBASE III, dBASE IV, Microsoft FoxPro и Paradox, осуществляется с помощью метода последовательного доступа (ISAM - Indexed Sequential Access Method). Это наиболее простой вариант технологии ODBC, о чем более подробно речь пойдет в [главе 8](#). Таким же образом может осуществляться использование текстовых файлов или электронных таблиц формата Microsoft Excel и Lotus 1-2-3.
- Данные, хранящиеся на сервере, для доступа к ним требуется несколько более сложная технология с использованием отдельных драйверов ODBC. Чтобы создать приложение для управления данными в архитектуре клиент-сервер, из программы на сервер необходимо передать команды SQL, впрочем, об этом вы прочитаете чуть позже, добравшись до [главы 8](#).

Объекты для доступа к данным имеют иерархическую организацию, в которой большинство классов относится к коллекциям классов, которые, в свою очередь, принадлежат другому классу выше в иерархии. Полная структура объектов для доступа к данным приведена на рис. 6.15.

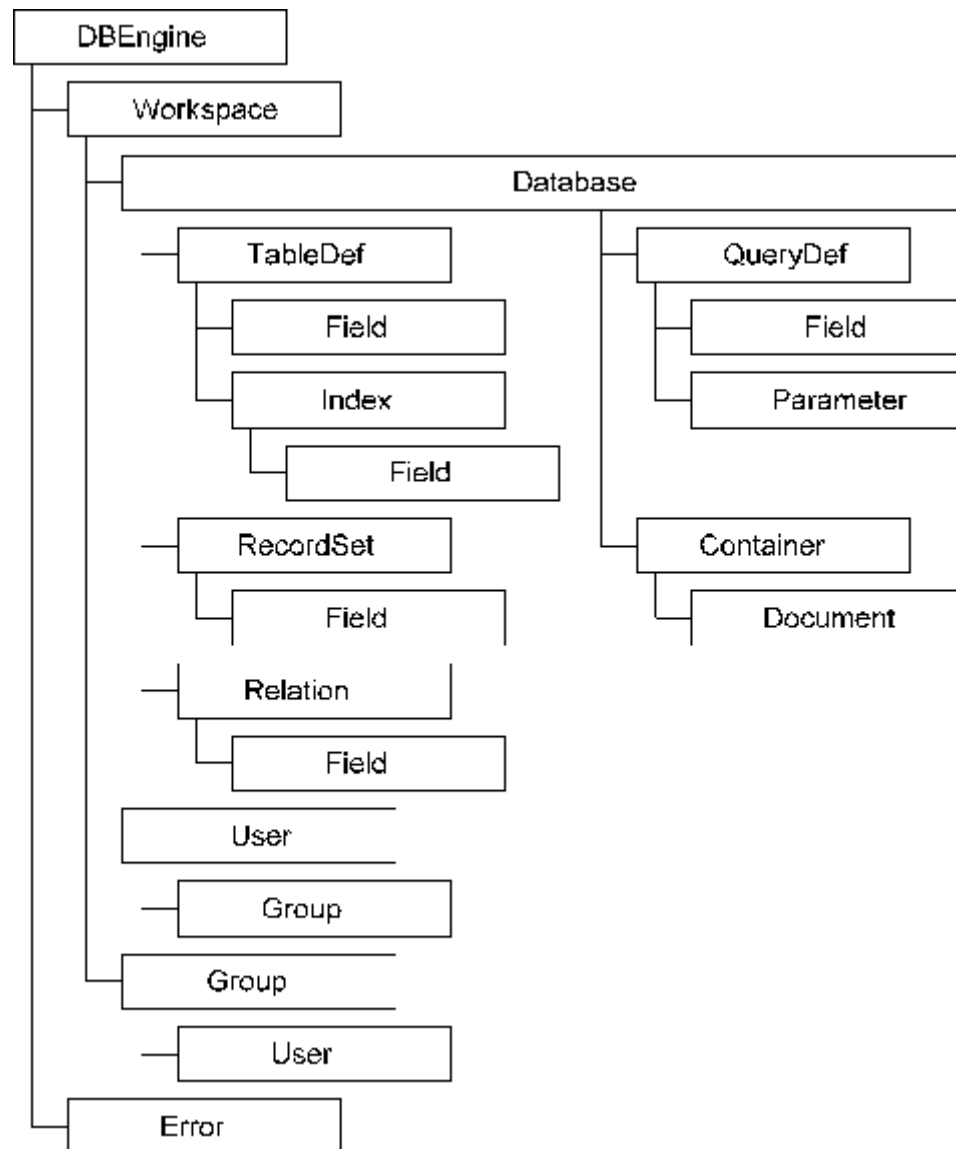


Рис. 6.15. Иерархия объектов для доступа к данным в Visual Basic

Мы поместили этот рисунок не для того, чтобы напугать вас мудреными словами "иерархия объектов". Этот способ хранения объектов реализуется с помощью специального вида объекта, который называется коллекцией. Единственная цель объекта коллекции состоит в том, чтобы содержать другие объекты. Объекты, содержащиеся в данной коллекции, всегда относятся к одному и тому же виду. Например, в коллекции **Indexes** могут содержаться только объекты **Index**.

Обратите внимание, что элементы иерархии объектов DAO на самом деле являются классами. Напомним, что класс - это не реальный объект, который может выполнять в программе какие-то действия, а только его прообраз. На основе определенного класса в программе мы создаем объект, который тут же готов выполнять предусмотренные в классе функции. Например, после такой строки в программе:

```
Dim oWksp As Workspace
```

создается объект на основе класса **Workspace**, для ссылки на который используется переменная **oWksp**.

Так как главной темой обсуждения в этом параграфе является поведение объектов, которые создаются в программе для обеспечения заданной функциональности, термин "объект" используется чаще, чем "класс". Только не забудьте, что в строгой формулировке, например, термин "объект **Database**" означает "объект класса **Database**".

Как вы можете видеть на рис. 6.15, большинство объектов доступа к данным представляется и

как объект, и как коллекция. На самом вершине иерархии - процессор данных, который представляется объектом **DBEngine**. Это единственный объект доступа к данным, который не может быть включен в какой-то другой объект. Объект **DBEngine** имеет коллекцию **Workspaces** (имя коллекции всегда имеет множественное число от имени содержащихся объектов), которая в свою очередь, может содержать один или большее количество объектов **Workspace**. Каждый объект **Workspace** имеет коллекцию **Databases**, которая содержит один или большее количество объектов **Database** и т. д.

Доступ к отдельным объектам коллекции может быть осуществлен по его номеру. Нумерация объектов в коллекции начинается с нуля. Например, на первый объект **TableDef** в базе данных с именем **MyDatabase** можно сослаться как **MyDatabase.TableDefs(0)**. На второй объект **TableDef** можно сослаться как **MyDatabase.TableDefs(1)** и т. д.

Объекты в иерархии должны указываться с полным перечнем "пути" от самого верхнего объекта, например:

```
DBEngine.Workspaces(0).Databases(0).TableDefs(0).Fields_ "Customer")
```

Как видно из этого примера, некоторые коллекции, помимо цифрового индекса, для ссылки на конкретный объект допускают ссылку по имени. Например, мы ссылаемся на объект коллекции **Fields** по значению свойства **Name** объекта **Field**.

В этом случае, при явной ссылке на объект, в отличие от ссылки по индексу, предпочтительнее использовать разделитель **!** (восклицательный знак). Таким образом, следующие выражения эквивалентны:

```
MyTableDef.Fields("Customer")
MyTableDef.Fields!Customer
```

Большинство объектов для доступа к данным имеют коллекции по умолчанию. Это позволяет использовать в программе более простой код, так как в этом случае не требуется явного указания имени коллекции при ссылке на объект из коллекции по умолчанию. Например, коллекцией по умолчанию для объекта **Recordset** будет являться коллекция **Fields**. Для получения значения поля **Customer** вы можете написать:

```
Cust = MyRecordset!Customer    что короче, чем
Cust = MyRecordset.Fields!Customer
```

Использование заданных по умолчанию коллекций упрощает код в прикладной программе, потому что при использовании вложенных ссылок на объекты заданные по умолчанию коллекции и составляют основную иерархию "пути" до объектов.

Основная идея объектно-ориентированного программирования заключается в том, что данные и процедуры, относящиеся к отдельному объекту, сохраняются все вместе внутри объекта. В **Visual Basic**, как и в других объектно-ориентированных языках программирования, данные, относящиеся к объекту (установки и атрибуты), называются свойствами, в то время как процедуры, с помощью которых выполняются какие-либо действия над объектами, называются методами.

Таким образом, написание программы с использованием объектов для доступа к данным заключается в создании переменной, которая будет выполнять функции ссылки на объект, а затем в манипулировании объектом путем выполнения соответствующих методов и установки требуемых значений свойств. В качестве примера давайте рассмотрим следующий фрагмент программы:

```
* Определяем переменные для ссылки на объекты
Dim MyDB As Database, MyWS As Workspace, MyRS As Recordset
* Открываем таблицу Clients в БД ACCOUNTS.MDB
Set MyWS = DBEngine.Workspaces(0)
Set MyDB = MyWS.OpenDatabase("ACCOUNTS.MDB")
Set MyRS = MyDB.OpenRecordSet("Clients")
* Устанавливаем порядок вывода записей
MyRS.Index = "ClientID"
```

Для открытия базы данных мы используем метод **OpenDatabase** объекта **Work-space** и привязываем к ней переменную **MyDB** с целью обеспечения дальнейших ссылок. Обеспечить доступ к записям таблицы **Clients** призван метод **OpenRecordset**. Требуемый логический порядок вывода записей мы устанавливаем с помощью свойства **Index**, которому присваиваем значение, соответствующее имени нужного индекса - **ClientID**.

Для большинства объектов доступа к данным программист может задать новые свойства, не предусмотренные стандартно в языке. Эта возможность позволяет гибко настраивать объекты в пользовательской программе и как угодно расширять перечень данных, сохраняемых вместе с объектом.

Теперь пришло время остановиться на каждом из объектов для доступа к данным, чтобы вы смогли получить представление о тех возможностях, которые могут быть реализованы с их помощью в прикладной программе. В табл. 6.4 приведен список свойств, а в табл. 6.5 - список методов, которые можно использовать для объектов каждого вида. В дальнейшем, при описании методов работы с данными, мы дадим более подробное описание перечисленных свойств и методов, но если вы хотя бы немного помните, что делали на уроках английского языка в школе, то уже сейчас сможете сделать некоторые выводы, изучив названия приведенных в таблицах свойств и методов.

DBEngine

Этот объект верхнего уровня ассоциируется с процессором данных. Он устанавливает системные параметры процессора данных и обеспечивает возможность работы с БД за счет автоматического создания объекта **Workspace** с номером 0. Коллекцией по умолчанию для этого объекта является **Workspaces**.

Workspace

Используется для поддержки транзакций, является контейнером для открытой БД и обеспечивает секретность работы с данными. Объект по умолчанию коллекции **Workspaces** - **Workspaces(0)** создается автоматически, как только в программе задается первая ссылка на объекты для доступа к данным. Этот объект может быть инициализирован с помощью свойств **Username** и **Password** перед активизацией объекта **DBEngine**. Коллекцией по умолчанию для этого объекта будет являться коллекция **Databases**.

Database

Этот объект корреспондируется с базой данных **Visual Basic**, внешней БД или соединением **ODBC**. Он используется для определения таблиц, связей и запросов к БД, а также для открытия объекта **Recordset**. Коллекцией по умолчанию является **TableDefs**.

TableDef

Каждый объект **TableDef** в коллекции **TableDefs** описывает соответствующую таблицу в текущей БД или присоединенную таблицу во внешней БД. В последнем случае с помощью этого объекта мы не можем изменить описание внешней таблицы. Коллекцией по умолчанию является **Fields**.

QueryDef

Этот объект является описанием сохраняемого запроса, который представляет собой объектный код с операторами **SQL**. С помощью этого объекта мы можем просмотреть и отредактировать при необходимости код, сохраняемый в запросе, установить его параметры и выполнить запрос. Коллекцией по умолчанию является **Fields**.

Recordset

Представляет собой курсор, который используется для отображения данных из таблицы БД или результата запроса. Курсор запоминает необходимые данные в виде набора записей в буфере и обеспечивает перемещение по этим записям с помощью методов **Move**, **Seek** и **Find**, выделяя текущую, что позволяет просматривать, обновлять или удалять необходимые данные. Этот объект является временным, и как только он закрывается, удаляется из коллекции, а все ассоциированные с ним данные из памяти. Коллекцией по умолчанию является **Fields**.

Field

Этот объект ассоциируется с колонкой данных, имеющих одинаковый тип и свойства. Коллекция объектов **Field** представляет собой запись в курсоре объекта **Recordset**. Данные в курсоре могут быть прочитаны и изменены с помощью свойства **Value** объекта **Field**. Как только указатель записи в курсоре перемещается на новую запись, все объекты **Field** в коллекции автоматически обновляются новыми значениями данных.

Index

В этом объекте хранится индекс, относящийся к объектам **TableDef** или **Recordset**, основанному на таблице. Нужный индекс может быть установлен с помощью свойства **Index**.

Parameter

Запоминает параметры для параметрического запроса. Коллекция **Parameters** объекта **QueryDef** позволяет получить или установить параметры для выполняемого запроса.

User

Используется для описания и поддержки условий доступа пользователей к информации, хранящейся в БД. Объект **DBEngine** поддерживает коллекцию пользователей. Добавление или удаление членов коллекции **Users** соответственно создает или стирает бюджеты пользователей. Каждый объект **User** создается с именем и паролем. Доступ к таким объектам, как **TableDef** или **QueryDef**, с помощью этого объекта может быть назначен индивидуально для конкретного пользователя.

Group

Представляет собой коллекцию пользователей с одинаковыми правами доступа к данным. Объект **DBEngine** поддерживает коллекцию групп пользователей. Каждый пользователь в группе наследует те права доступа, которые предоставлены группе. Это облегчает управление доступом к данным для нескольких пользователей.

Relation

Этот объект используется для хранения данных о связях между полями двух объектов **TableDef**. Каждый объект **Database** имеет единственную коллекцию объектов **Relation**. Процессор данных использует информацию о связях для определения возможности обновления и удаления данных без потери целостности данных в БД.

Property

Запоминает значения свойств какого-либо объекта. При этом сохраняются как встроенные в язык свойства, так и добавленные для данного объекта программистом. Свои свойства можно описать для следующих объектов:

- **Database**
- **TableDef**
- **QueryDef**
- **TableDef.Index**
- **TableDef.Field**
- **QueryDef.Field**

Container

Этот объект используется для соединения с объектами **Document** и перечисления объектов, размещаемых в БД, включая объекты, определяемые пользовательской программой. Каждый объект **Database** может иметь только одну коллекцию объектов **Container**.

Таблица 6.5. Методы объектов для доступа к данным

[illegible]

CreateProperty	+	++	++	++	++	++
CreateQueryDef	+					
CreateRelation	+					
CreateReplica	+					
CreateTabledef	+					
CreateUser	+					++
CreateWorkspace	+					
Delete					+	
Edit					+	
Execute		+		++		
FieldSize						+
FillCache					+	
FindFirst					+	
FindLast					+	
FindNext					+	
FindPrevious					+	
GetChunk						+
GetRows					+	
Idle	+					
Move					+	
MoveFirst					+	
MoveLast					+	
MoveNext					+	
MovePrevious					+	
MoveReplica		+				
NewPassword						
OpenDatabase						
OpenRecordset		+	++	++	++	++
RefreshLink			+			
RegisterDatabase	+					
RepairDatabase	+					
Requery					+	
Rollback		+				
Seek					+	
Synchronize		+				
Update					+	

Теперь, когда вы получили представление о концепции использования объектов доступа к данным и процессора данных для управления БД, мы можем обсудить, как использовать этот инструмент для создания базы данных.

В Visual Basic существует несколько способов создания БД.

На этапе разработки прикладной программы можно использовать самый простой способ - с помощью Data Manager. После запуска Visual Basic в меню Add-Ins выберите команду Data Manager. В его окне в меню File выберите команду New Database. Задайте имя для создаваемой БД и выберите папку, в которой она будет располагаться. В окне Data Manager появится окно с именем новой БД, как это видно на рис. 6.16. С его помощью вы также можете внести в структуру БД необходимые изменения, добавить или отредактировать хранящиеся в ней данные.

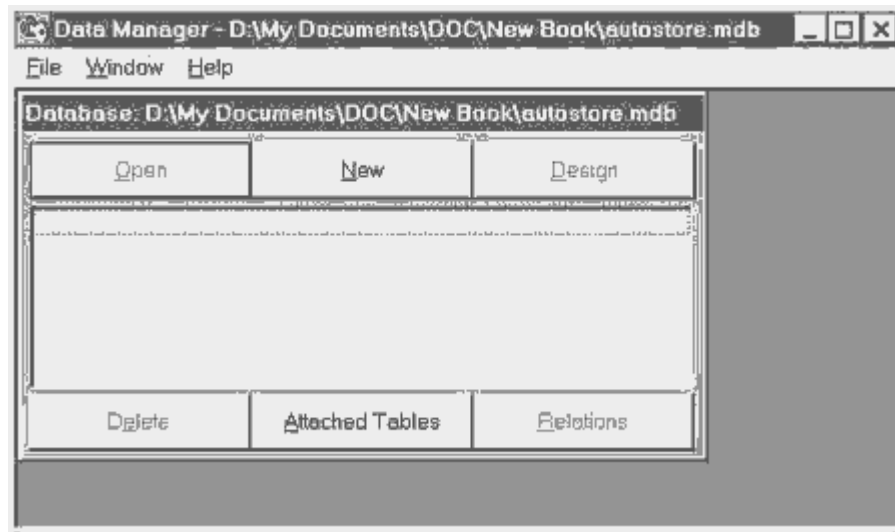


Рис. 6.16. Создание базы данных с помощью Data Manager

Второй способ создания БД основывается на использовании в Visual Basic формата хранения данных Microsoft Access. Тем самым любая БД, созданная в СУБД Access 7.0, может быть "полнокровно" использована в программе, написанной на Visual Basic.

Естественно, посредством технологии ODBC мы можем создать, а в дальнейшем управлять из программы Visual Basic базой данных, созданной в любой другой СУБД при наличии соответствующего драйвера.

Перечисленные варианты, как правило, не приемлемы, если в соответствии с заданной функциональностью необходимо создание БД в процессе работы прикладной программы. Поэтому использование объектов доступа к данным для создания БД обеспечивает максимальную гибкость и самые широкие возможности в работе. Перед тем как использовать объекты для доступа к данным, не забудьте убедиться, что установлена ссылка на соответствующую библиотеку. В меню *Tools* при задании команды *References* в появляющемся диалоговом окне должна быть помечена библиотека Microsoft DAO 3.0 Object Library.

Для создания новой БД в меню *File* Visual Basic выберем команду **New Project**. Назовем проект Create_DB. Создадим в проекте программный модуль путем выполнения команды **Module** в меню *Insert*. После в этом же меню выполним команду **Procedure**. В появившемся диалоговом окне напомним имя создаваемой процедуры - **Main**, для типа процедуры выберем **Sub**, а для диапазона действия - **Public**. После нажатия кнопки **OK** появится шаблон, готовый для написания программного кода, как это показано на рис. 6.17. Далее необходимо выполнить следующие действия.



Рис. 6.17. Шаблон для написания программы

1. Используйте оператор **Dim** для создания новых переменных, с помощью которых будет выполняться ссылка на соответствующий объект, включенный в БД. Дополнительно к объектам DBEEngine и Workspace, которые определяют рабочую среду, необходимо иметь:

- Один объект Database.
- Один объект TableDef для каждой таблицы.
- Один объект Field для каждого поля каждой таблицы.
- Один объект Index для каждого индекса таблицы.

Таким образом, для создания фрагмента БД Autostore, состоящего из двух связанных таблиц Customer и Account, мы должны определить следующие переменные:

```
Public Sub Main()
' Определяем переменные
Dim oAutoDB As Database, oAutoWs As Workspace
Dim oCustomerTd As TableDef, oAccountTd As TableDef
Dim oCustomerFlds(8) As Field, oAccountFlds(7) As Field
Dim oCustomerIdx As Index, oAccountIdx(2) As Index
Dim oAutoRel As Relation
Dim oIndexFld(3), oRelFld As Field
```

Для непосредственного создания БД используйте метод CreateDatabase объекта Workspace:

```
Set oDBVar = [oWSVar].CreateDatabase(cDataBaseName,
Locale [, Options])
```

где

- **oDBVar** - имя переменной для ссылки на БД;
- **oWSVar** - имя переменной для ссылки на объект Workspace;
- **cDataBaseName** - имя создаваемой БД, которому может предшествовать указание пути доступа к файлам;
- **Locale** - последовательность расположения символов в таблицах БД при сортировке или индексировании данных. Для России наиболее подходящим может быть использование последовательности, соответствующей стандартной таблице расположения символов, которая устанавливается заданием константы **dbLangGeneral**, или последовательности, соответствующей русскому алфавиту. Для этого надо использовать константу **dbLangCyrillic**.
- **Options** - задает дополнительные параметры для БД. Например, можно использовать константу **dbEncrypt** для шифрования данных в создаваемой БД.

Для создания БД Autostore используем следующий код:

```
Set oAutoWs = DBEngine.Workspaces(0)
Set oAutoDB = oAutoWs.CreateDatabase("AUTOSTORE.MDB", _ dbLangGeneral)
```

Чтобы создаваемые файлы при записи на диск оказались в нужном месте, можно использовать операторы **ChDrive** и **ChDir**.

3. Используйте метод CreateTableDef объекта Database для создания таблиц в БД:

```
Set oTDVar = oDBVar.CreateTableDef([cName
[, Attributes [, Source [,]]]])
```

где

- **oTDVar** - имя переменной для ссылки на объект TableDef.
- **oDBVar** - имя переменной для ссылки на объект Database.
- **cName** - имя создаваемой таблицы. Оно должно начинаться с буквы и иметь не более 40 символов. В имени таблицы не должно использоваться знаков пунктуации и пробелов.
- **Attributes** - устанавливает дополнительные характеристики для создаваемой таблицы, в основном используемые для внешних источников данных.
- **Source** - указывает имя таблицы внешней БД, которая будет являться источником данных.
- **Connect** - определяет характеристики соединения в случае использования внешнего источника данных.

Создадим две таблицы в БД Auto_Store, используя следующие строки:

```
Set oCustomerTd = oAutoDB.CreateTableDef("Customer")
```

```
Set oAccountTd = oAutoDB.CreateTableDef("Account")
```

4. Для создания полей в таблице используйте метод `CreateField` объекта `TableDef`:

```
Set oFVar = oTDVar.CreateField([cName [, Type [, Size]])
```

где

- **oFVar** - имя переменной для ссылки на объект `Field`.
- **oTDVar** - имя переменной для ссылки на объект `TableDef`.
- **cName** - имя создаваемого поля. Правила задания имени поля такие же, как для имен таблиц.
- **Type** - тип создаваемого поля. В `Visual Basic` вы можете использовать следующие константы для указания типа поля:
 - **dbDate** - дата и время;
 - **dbText** - символьное;
 - **dbMemo** - поле примечаний;
 - **dbBoolean** - логическое (значения Yes/No);
 - **dbInteger** - целое число (2 байта);
 - **dbLong** - целое число двойной точности (4 байта);
 - **dbCurrency** - денежное выражение;
 - **dbSingle** - числовое;
 - **dbDouble** - числовое с плавающей точкой двойной точности;
 - **dbByte** - целое положительное число;
 - **dbLongBinary** - символьная строка для ссылки на OLE-объект.
- **Size** - ширина создаваемого поля. Этот параметр необходимо указывать только для символьных полей (1 - 255).

Проиллюстрируем использование этого метода на примере создания нескольких полей для таблицы `Customer`:

```
Set oCustomerFlds(0) = oCustomerTd.CreateField("KEY_CUSTOMER", dbLong)
` Установим для этого поля автоматическое приращение
` значения с помощью свойства Attributes объекта Field
oCustomerFlds(0).Attributes = dbAutoIncrField
Set oCustomerFlds(1) =_ oCustomerTd.CreateField("NAME_CUSTOMER", dbText, 100)
...
```

5. Для создания индексов используйте метод `CreateIndex` объекта `TableDef`:

```
Set oIVar = oTDVar.CreateIndex([cName])
```

где

- **oIVar** - имя переменной для ссылки на объект `Index`.
- **oTDVar** - имя переменной для ссылки на объект `TableDef`.
- **cName** - имя индекса. Правила его задания такие же, как для имени таблицы.

Необходимые характеристики для создаваемого индекса можно задать с помощью соответствующих свойств объекта `Index` (см. табл. 6.4). Для таблицы `Customer` создание первичного индекса может быть выполнено следующим образом:

```
Set oCustomerIdx = oCustomerTd.CreateIndex("CUSTOMER_ID")
oCustomerIdx.Primary = True
oCustomerIdx.Unique = True
```

Теперь следует с помощью метода `CreateField` создать поле, для того чтобы указать, какое поле будет использовано в качестве ключевого, как это показано в следующем примере:

```
Set oIndexFld(0) =_ oCustomerIdx.CreateField("KEY_CUSTOMER")
```

6. Установите отношение между таблицами. Для этого используйте метод `CreateRelation` объекта `Database`:

```
Set oRelVar = oDBVar.CreateRelation([cName
[, cParentTable [,cChildTable
[, Attributes]]]])
```

где

- **oRelVar** - имя переменной для ссылки на объект `Relation`.
- **oDBVar** - имя переменной для ссылки на объект `Database`.
- **cName** - имя создаваемого объекта. Правила его составления такие же, как для таблицы.
- **cParentTable** - имя родительской таблицы в создаваемом отношении.
- **cChildTable** - имя дочерней таблицы в создаваемом отношении.
- **Attributes** - задает тип создаваемого отношения за счет использования следующих констант:
 - **dbRelationUnique** - отношение "один к одному".
 - **dbRelationDontEnforce** - в отношении не поддерживается целостная ссылочность.
 - **dbRelationInherited** - отношение существует не в текущей БД между двумя присоединенными таблицами.
 - **dbRelationLeft** - в родительской таблице могут оставаться записи, не имеющие соответствующих записей в дочерней таблице.
 - **dbRelationRight** - в дочерней таблице могут оставаться записи, не имеющие соответствующих записей в родительской таблице.
 - **dbRelationUpdateCascade** - в отношении будет поддерживаться каскадное обновление.
 - **dbRelationDeleteCascade** - в отношении будет поддерживаться каскадное удаление.

После создания отношения необходимо создать поле для отношения в родительской таблице и указать соответствующее ему поле в дочерней таблице, как это показано в следующем примере:

```
' Создание связи между таблицами "Один ко многим"
Set oAutoRel = oAutoDB.CreateRelation("CustToAcc",_ "Customer", "Account")
Set oRelFld = oAutoRel.CreateField("KEY_CUSTOMER")
oRelFld.ForeignName = "KEY_CUSTOMER"
```

7. Добавьте созданные поля и индексы в соответствующие таблицы, а таблицы и отношение - в БД с помощью метода `Append`. Обратите внимание, что после добавления объектов в соответствующие коллекции большинство их свойств не может быть изменено. Для изменения свойств в этом случае вам придется сначала удалить соответствующий объект с помощью метода `Delete`, а затем создать его заново с требуемыми значениями свойств и после этого добавить в коллекцию.

Collection.Append oVar

где

- **Collection** - имя коллекции, в которую добавляется объект;
- **oVar** - имя переменной для ссылки на добавляемый объект.

Для нашего примера фрагмент кода, обеспечивающий добавление объектов в коллекции, будет выглядеть следующим образом:

```
` Добавляем поля в таблицы
oCustomerTd.Fields.Append oCustomerFlds(0)
oCustomerTd.Fields.Append oCustomerFlds(1)
...
` Добавляем поля в индексы
oCustomerIdx.Fields.Append oIndexFld(0)
oAccountIdx(0).Fields.Append oIndexFld(1)
oAccountIdx(1).Fields.Append oIndexFld(2)
```



```

` Добавляем поля в отношение
oAutoRel.Fields.Append oRelFld
` Добавляем индексы в таблицы
oCustomerTd.Indexes.Append oCustomerIdx
oAccountTd.Indexes.Append oAccountIdx(0)
oAccountTd.Indexes.Append oAccountIdx(1)
` Добавляем таблицы в БД
oAutoDB.TableDefs.Append oCustomerTd
oAutoDB.TableDefs.Append oAccountTd
` Добавляем отношение в БД
oAutoDB.Relations.Append oAutoRel

```

Для ясности в примере мы использовали различные переменные для каждого объекта доступа к данным. На практике обычно для сокращения объема кода используют одну и ту же переменную сначала для создания объекта, а после его добавления в коллекцию - для ссылки на следующий объект.

После создания БД вы можете использовать объекты доступа к данным для программного изменения ее структуры. Модификация БД очень похожа на те действия, которые мы выполняли при ее создании. В большинстве случаев для добавления объектов достаточно использовать те же самые методы **Create** и **Append**. Вы также можете добавлять в БД новые таблицы, добавляя новые объекты **TableDef** или новые поля и индексы, добавляя новые объекты **Field** и **Index** в существующие таблицы. Если вам больше нравится SQL, используйте для модификации БД операторы этого языка.

Давайте посмотрим на следующий простейший пример добавления таблицы в существующую БД с помощью объектов DAO:

```

Dim oDB As Database
' Переменная для нового объекта TableDef
Dim oNewTd As TableDef
Dim oNewFld As Field ' Переменная для нового объекта Field
' Открываем БД
Set oDB = _DBEngine.Workspaces(0).OpenDatabase("AUTOSTORE.MDB")
Set oNewTd = oDB.CreateTableDef("Новая таблица")
Set oNewFld = oNewTd.CreateField("Новое поле", dbInteger)
' Добавляем новое поле в таблицу
oNewTd.Fields.Append oNewFld
oDB.TableDefs.Append oNewTd ' Добавляем таблицу в БД
oDB.Close ` Закрываем БД

```

Естественно, существующие в БД объекты могут быть удалены. Мы можем использовать метод **Delete** для удаления таблицы, добавленной в БД в предыдущем примере:

```
oDB.TableDefs.Delete "Новая таблица"
```

Следует иметь в виду, что при удалении индексного поля сначала следует удалить сам индекс и связанные с ним объекты **Relation**. Лишь после этого вы сможете удалить объекты **Field** или **TableDef**, являющиеся составными частями отношения.

На прилагаемой к книге дискете помещен полный код программы для создания БД **Auto_Store** в Visual Basic.

6.4. MS SQL Server

Создать базу данных в MS SQL Server можно несколькими путями: визуально, находясь в Microsoft SQL Enterprise Manager, программно, посредством редактора Microsoft ISQL/w, программно посредством технологии SQL pass-through из клиентского приложения и путем наращивания (upsizing) локальной БД.

В этом параграфе мы опишем создание базы данных как программным путем, так и с помощью процесса наращивания.

Программный путь создания БД посредством одной из составных частей MS SQL Server - редактора Microsoft ISQL/w - реализуется простым набором команд, которые мы можем тут же запускать на выполнение. Редактор ISQL/w имеет несколько страниц, на первой из которых мы можем набирать необходимые команды, как это показано на рис. 6.18. Набранную команду можно тут же запустить на выполнение, нажав кнопку с треугольником в верхней правой части окна. На вкладке **Results** вы можете просмотреть результат выполнения команды, если она

предусматривает вывод данных (рис. 6.19). Остальные вкладки обеспечивают возможность очень наглядного графического представления условий выполнения команды. Вы можете убедиться в этом, взглянув на рис. 6.20 и 6.21.

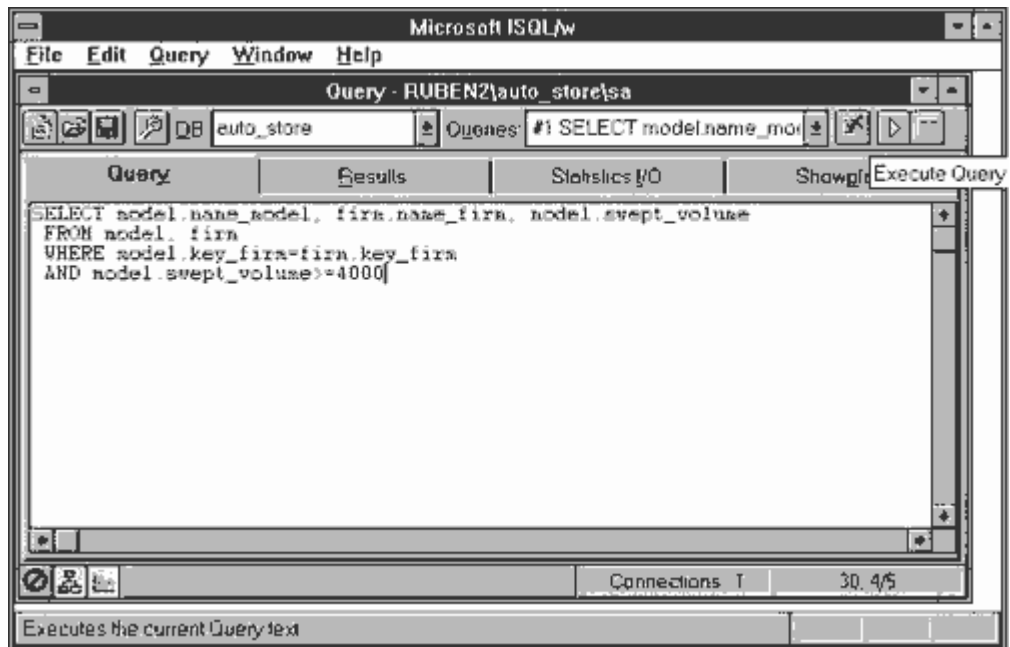


Рис. 6.18. Набор команды в редакторе SQL Server

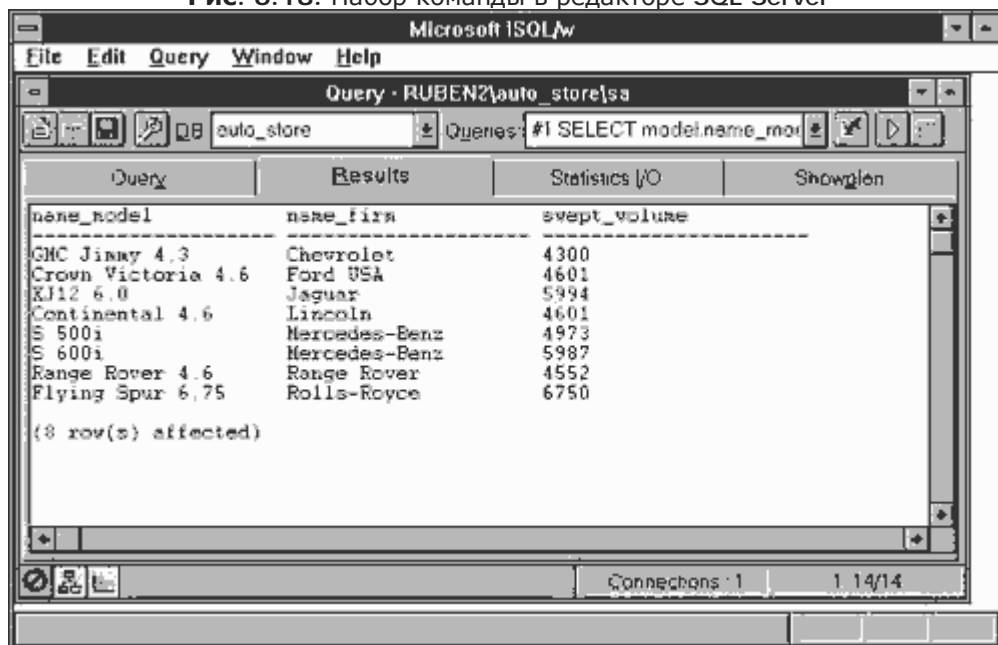


Рис. 6.19. Просмотр результатов выполнения команды в SQL Server

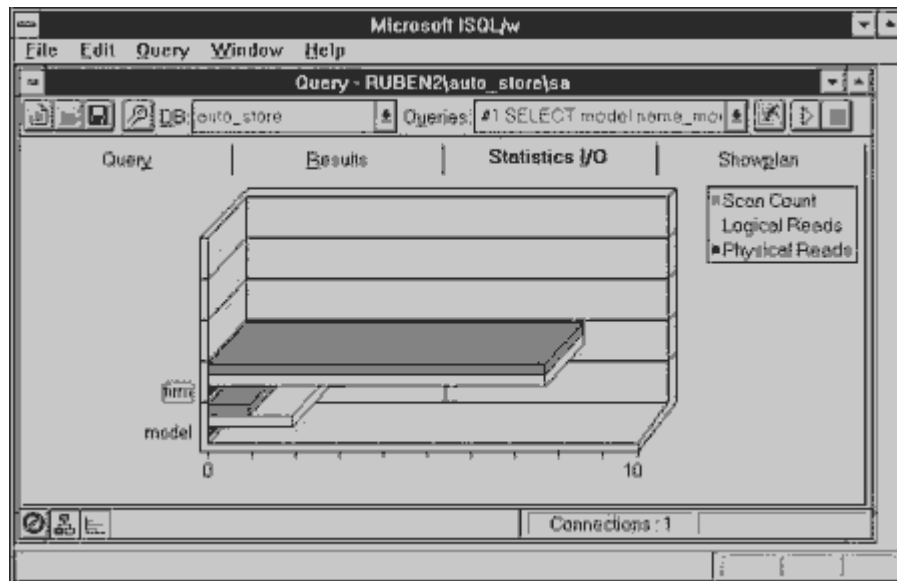


Рис. 6.20. Графическое представление выборки в SQL Server

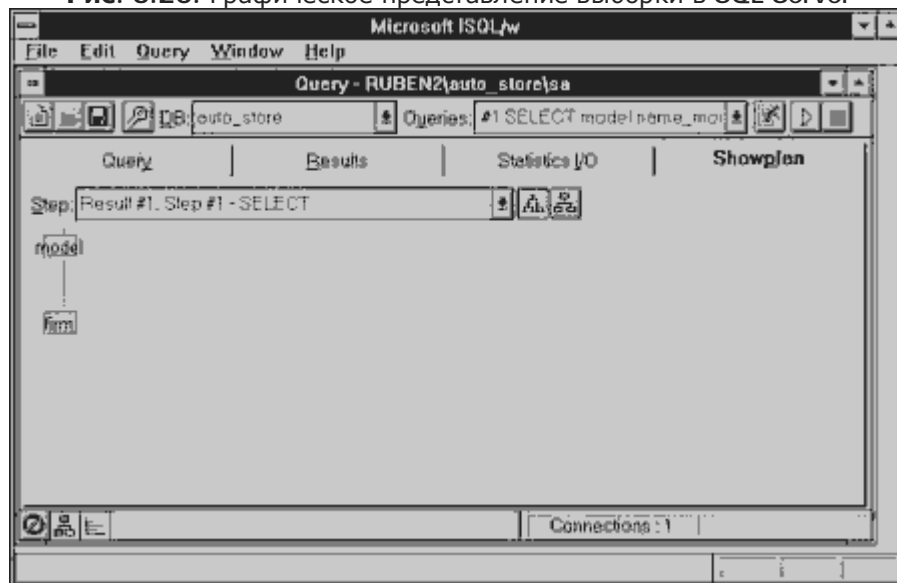


Рис. 6.21. Графическое представление отношения в выборке между таблицами в SQL Server

Прежде чем создать БД, необходимо решить, на каком из существующих устройств - Device - наша база данных будет храниться, либо создать подобное устройство.

Device - файл, который MS SQL Server использует для размещения одной или нескольких баз данных, заранее резервируя место на диске.

Размер Device указывается уже при его создании. При этом необходимо учитывать, что база данных может находиться на нескольких устройствах, то есть Device может хранить часть базы данных. Устройство создается следующей командой:

```
DISK INIT
NAME = cLogicalName,
PHYSNAME = cPhysicalName,
VDEVNO = nValue1,
SIZE = nValue2
[, VSTART = Address]
```

Описание опций этой команды приведено в следующей таблице:

Предложение	Описание
NAME	Название устройства, которое будет использоваться внутри SQL для обращения к нему. Не может превышать 30 символов.
PHYSNAME	Дисковый файл, в котором устройство будет храниться. Его название должно удовлетворять требованиям операционной системы.
VDEVNO	Уникальный номер устройства между 1 и 255.
SIZE	Количество 2-килобайтных блоков. Минимальное число - 500 блоков или 1 Мбайт.
VSTART	Смещение первой базы данных, которая будет храниться в устройстве. По умолчанию 0.

В качестве примера создадим устройство Rdev_lib, где SAMP_LIB.DAT - имя файла, который является хранилищем устройства:

DISK INIT

```
NAME = 'Rdev_lib',
PHYSNAME = 'C:\SQL_60\DATA\SAMP_LIB.DAT',
VDEVNO=1,
SIZE = 5120
```

На рис. 6.22 приводится результат создания устройства, который мы можем увидеть в File Manager Windows NT.

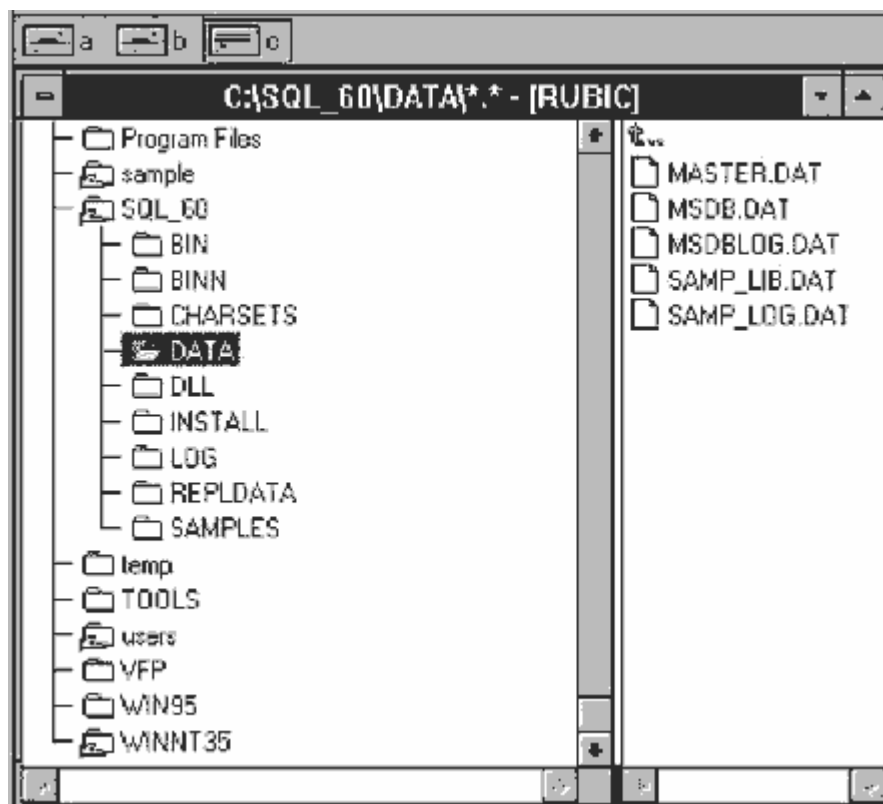


Рис. 6.22.

Далее обязательно необходимо создать устройство Rdev_log для журнала транзакций БД:

DISK INIT

```

NAME = 'Rdev_log',
PHYSNAME = 'C:\SQL_60\DATA\SAMP_LOG.DAT',
VDEVNO=2,
SIZE = 2048

```

На рис. 6.23 и 6.24 показаны характеристики вновь созданных устройств, а на рис. 6.25 - их отображение в Server Manager.

Теперь мы можем создать базу данных **Auto_Store**, которая будет размещаться на двух устройствах: **Rdev_lib** и **Rdev_log**, где числа 10 и 4 обозначают резервируемое пространство в мегабайтах. Синтаксис команды, с помощью которой создается база данных:

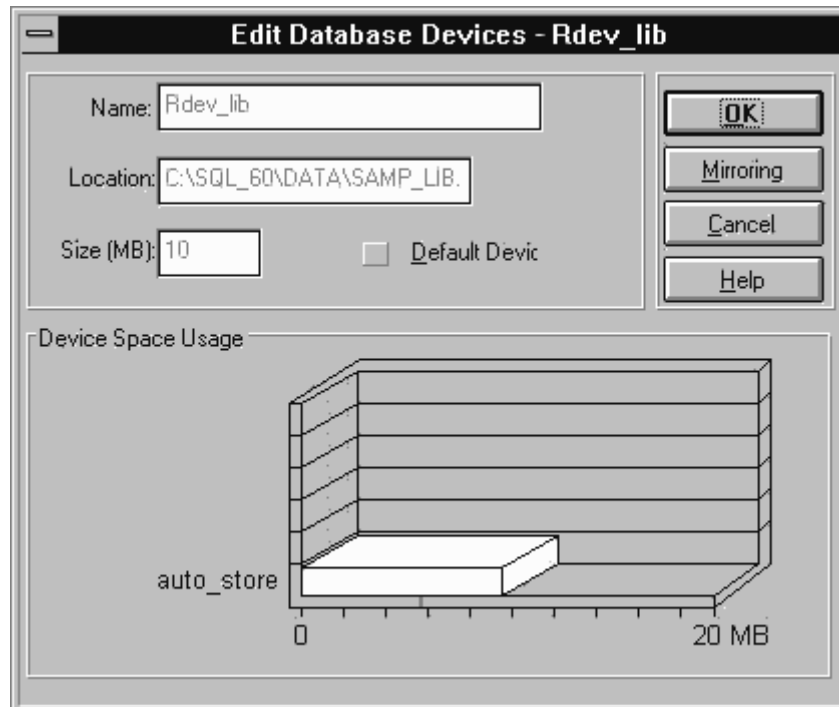


Рис. 6.23.

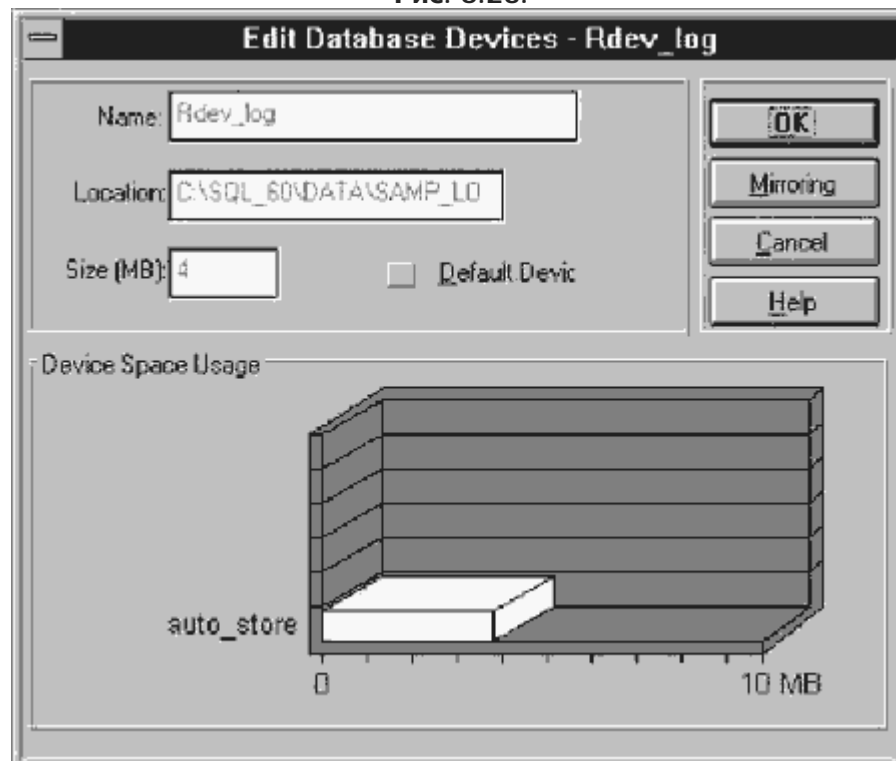


Рис. 6.24.

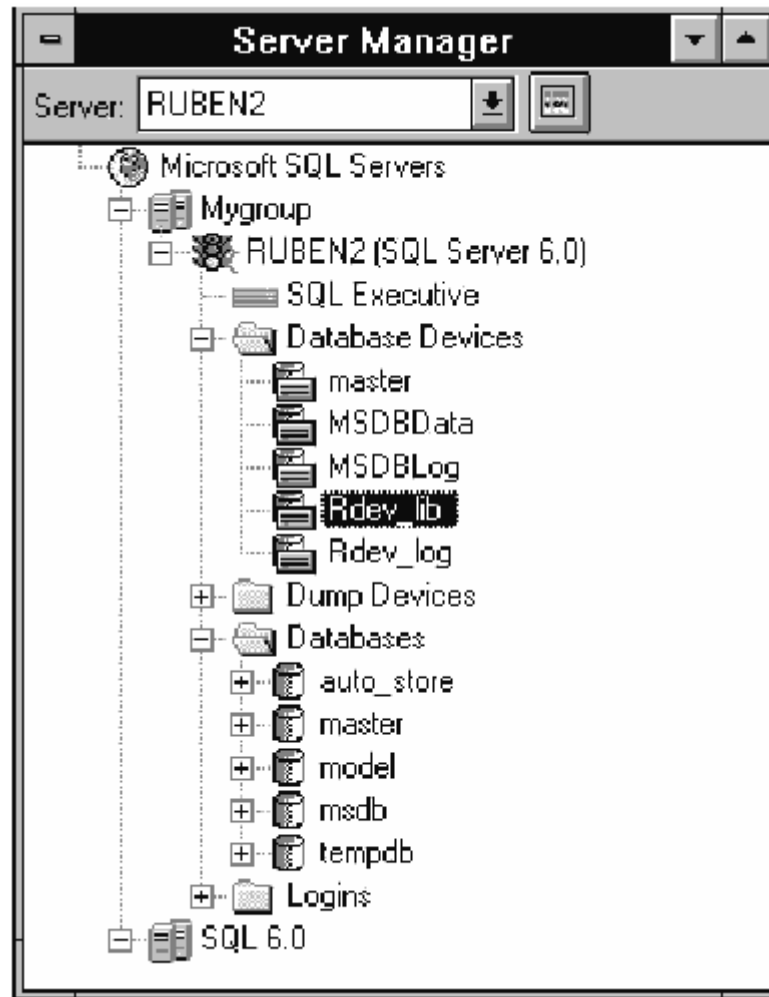


Рис. 6.25.

```
CREATE DATABASE DatabaseName
[ON {DEFAULT | DatabaseDevice} [= Size]
    [, DatabaseDevice [= Size]]...]
[LOG ON DatabaseDevice [= Size]
    [, DatabaseDevice [= Size]]...]
[FOR LOAD]
```

Аргумент **DatabaseName** указывает имя вновь создаваемой базы данных. Как мы уже упоминали и это видно из приведенного синтаксиса, вы можете распределить базу данных между несколькими устройствами, резервируя определенный размер на каждом устройстве. Если вы укажете ключевое слово **DEFAULT**, то база данных будет создана на устройстве по умолчанию, которое определено в таблице **Sysdevices**, то есть вы можете написать: **ON DEFAULT = 5**. Устройство, где будет храниться журнал транзакций базы данных, вы можете установить с помощью ключевого слова **LOG ON**. Можно указать более, чем одно устройство.

Опция **FOR LOAD** резервирует базу данных для перезагрузки предыдущей копии базы. Если вы указываете опцию **FOR LOAD**, никто не сможет ни случайно, ни преднамеренно редактировать базу данных между временем создания базы данных и ее загрузкой.

```
CREATE DATABASE Auto_Store ON Rdev_lib = 10 LOG ON Rdev_log = 4
```

База данных **Auto_Store** существует, но пока она совершенно пуста. Создадим в ней две таблицы: **Country** и **Firm**, связанные по полю **key_country**.

```
USE Auto_Store
```

Создаем таблицу **Country**:

```
CREATE TABLE Country (key_country smallint IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_country varchar(20) NOT NULL, times_ timestamp)
```

Создаем таблицу Firm:

```
CREATE TABLE firm (key_firm smallint IDENTITY(1,1)
PRIMARY KEY CLUSTERED, name_firm varchar(20) NOT NULL,
key_country smallint REFERENCES country(key_country),
times_timestamp)
```

На рис. 6.26 приведен результат выполнения команд **CREATE TABLE** и **ALTER TABLE** для таблицы Model, а на рис. 6.27 - список всех созданных таблиц в Server Manager.

Key	Identity	Column Name	Datatype	Size	Nulls	Default
✓	✓	key_model	smallint	2		
		name_model	varchar	20		
		key_firm	smallint	2	✓	(1)
		swept_volume	numeric	5,0	✓	(1)
		quantity_drum	numeric	2,0	✓	(1)
		capacity	numeric	5,1	✓	
		torque	numeric	5,1	✓	
		key_fuel_oil	smallint	2	✓	(1)
		top_speed	numeric	5,1	✓	
		starting	numeric	4,1	✓	
		key_tyre	smallint	2	✓	(1)
		key_body	smallint	2	✓	(1)
		quantity_door	numeric	1,0	✓	
		quantity_seat	numeric	2,0	✓	
		length	numeric	5,0	✓	
		width	numeric	4,0	✓	
		height	numeric	4,0	✓	

Рис. 6.26. Отображение данных для таблицы Model в Manage Tables

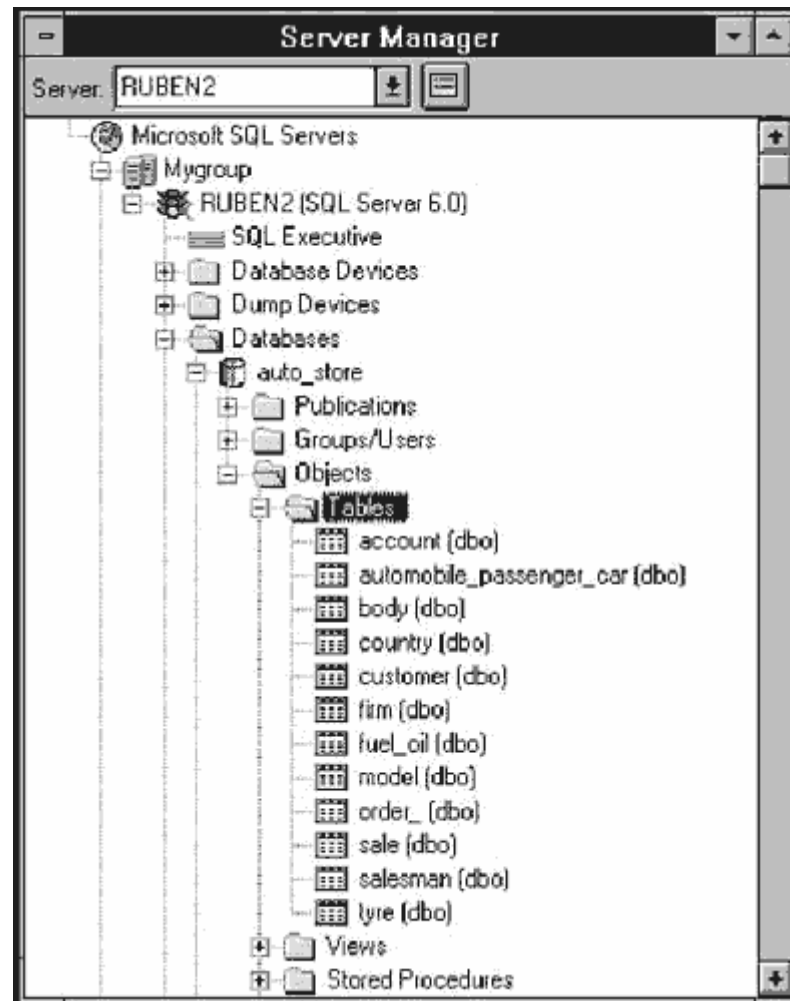


Рис. 6.27.

Приведем правила SQL Server для присвоения идентификаторов полей:

1. Названия должны быть длиной от 1 до 30 символов.
2. Первым символом может быть буква или знаки "_", "@", "#". Идентификаторы, которые начинаются с "@", зарезервированы для локальных переменных, а идентификаторы, которые начинаются с "#", зарезервированы для временных объектов.
3. Остальными символами могут быть буквы, цифры, знаки "\$" или ".".
4. Пробелы внутри названий не допускаются.
5. Названия полей должны быть уникальными для одного и того же пользователя в одной и той же таблице.
6. Названия колонок должны быть уникальными внутри одной таблицы.

Самое время сделать несколько замечаний относительно использования русского языка в названиях таблиц и полей. Естественно, что все рассматриваемые СУБД, работающие под управлением Windows 95, прекрасно поддерживают эту возможность. Оказывается, такой на порядок более серьезный продукт, как SQL Server, также позволяет создавать таблицы с русскими полями, более того, имя таблицы также может быть русским. Не возникает проблем и при наращивании БД (upsizing) с русскими названиями полей и таблиц в архитектуру клиент-сервер. Например:

```
CREATE TABLE Таблица_с_русскими_полями (код smallint IDENTITY(1,1)
PRIMARY KEY CLUSTERED, наименование varchar(20) NOT NULL, время timestamp)
```

На рис. 6.28 приведен результат выполнения данного примера.

После создания таблиц необходимо позаботиться о привилегиях доступа пользователей к той или иной таблице или полю. Привилегии доступа выделяются пользователям или группам посредством оператора **GRANT** и изымаются с помощью оператора **REVOKE**. Эти привилегии, как

правило, присваивает владельцу соответствующих объектов (он же - администратор базы данных).

Key	Identity	Column Name	Datatype	Size	Nulls	Default
	<input checked="" type="checkbox"/>	код	smallint	2		
		наименование	varchar	20		
		время	timestamp	8	<input checked="" type="checkbox"/>	

Рис. 6.28.

Применительно к таблицам и представлениям можно управлять следующими правами доступа:

- **SELECT** - право на выборку данных;
- **INSERT** - право на добавление данных;
- **DELETE** - право на удаление данных;
- **UPDATE** - право на обновление данных (можно указать определенные столбцы, разрешенные для обновления);
- **REFERENCES** - право на использование внешних ключей, ссылающихся на данную таблицу (можно указать определенные столбцы).

Допустим, что у нас в системе два пользователя, Karina и Lena, которые входят в одно подразделение Piter_group, и нам необходимо предоставить полномочия, указанные в табл. 6.6.

Таблица 6.6. Образец назначения привилегий доступа для пользователей

Таблицы	Поля	SELECT	INSERT	DELETE	UPDATE
Country	key_country		Karina	Karina	
	name_country	Karina, Lena	Karina	Karina	Karina, Lena
Firm	times_			Karina	Karina
	key_firm			Lena	Lena
	name_firm	Karina, Lena	Lena	Lena	Karina, Lena
	key_country			Lena	Lena
	times_			Lena	Lena

Для назначения указанных полномочий создаем группу Piter_group:

```
sp_addgroup piter_group
```

Последовательно создаем процедуры регистрации с именами login_lev1, login_lev2 и соответствующими им паролями lev1, lev2 для БД Auto_Store:

```
sp_addlogin login_lev1, lev1, auto_store
sp_addlogin login_lev2, lev2, auto_store
```

Созданные процедуры регистрации появляются в списке, который можно просмотреть в Server Manager, как это показано на рис. 6.29.

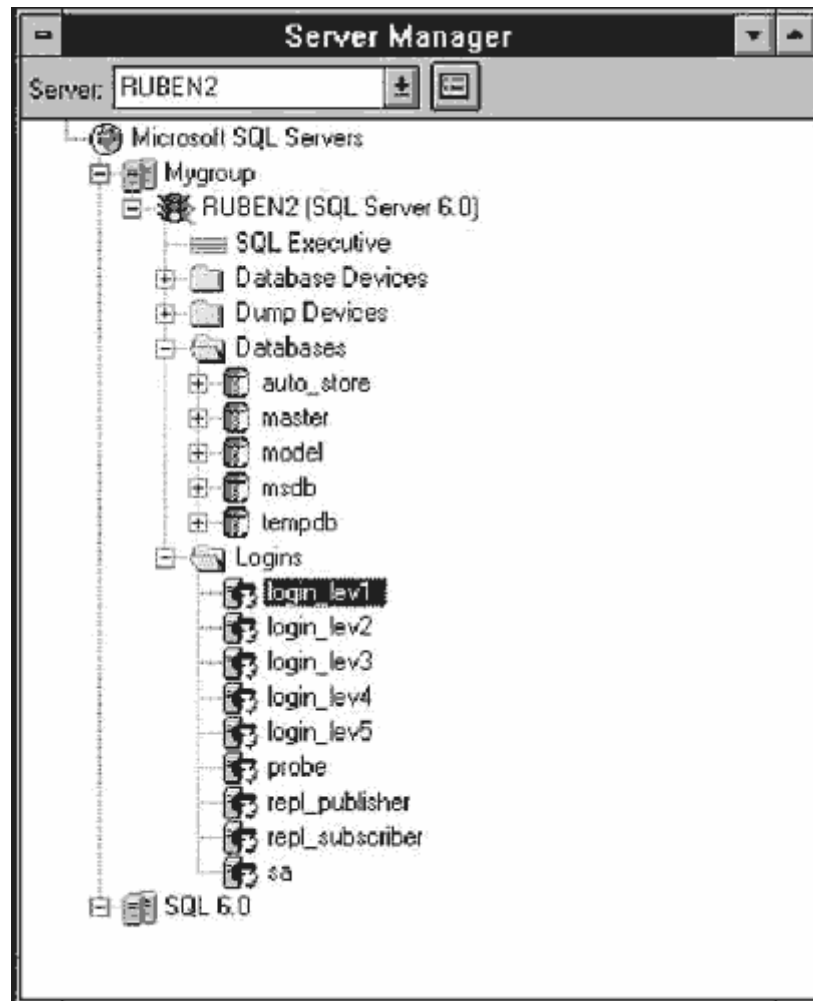


Рис. 6.29. Список Logins - результат выполнения хранимой процедуры `sp_addlogin`

Создаем пользователей с именами Karina и Lena для группы Piter_group:

```
sp_adduser login_lev1, karina, piter_group
sp_adduser login_lev2, lena, piter_group
```

Пользователей в новой группе вы можете увидеть в соответствующем списке **Server Manager**, как это показано на рис. 6.30. Появились новые пользователи, что является результатом хранимых процедур `sp_addgroup`, `sp_adduser`.

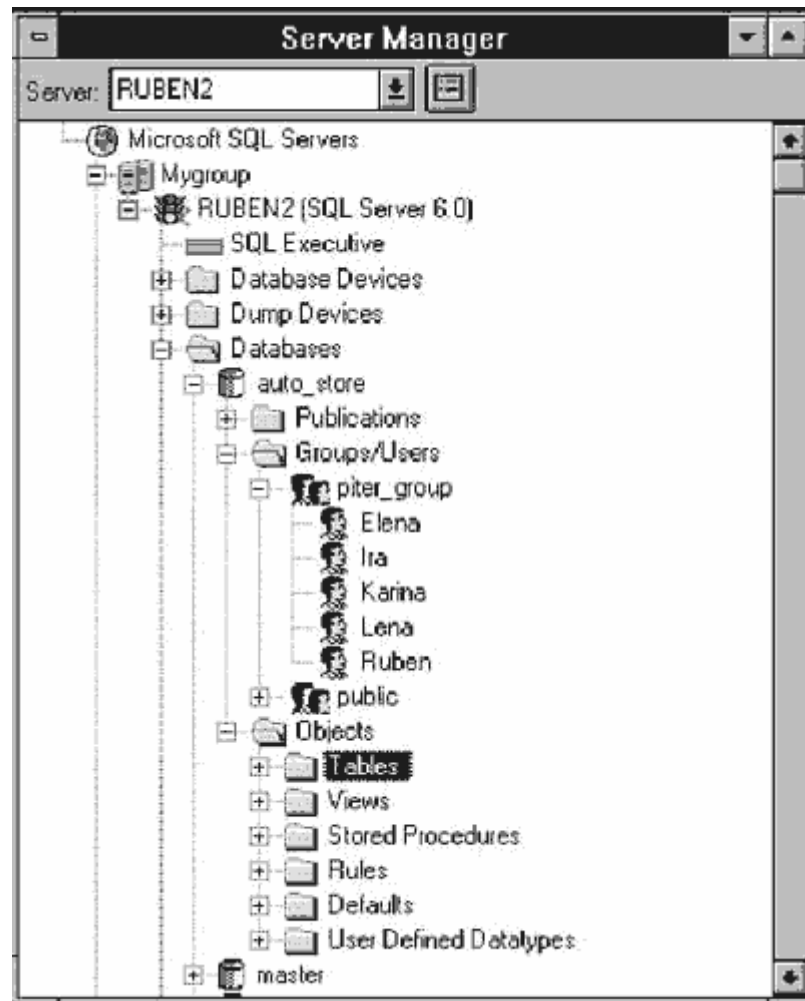


Рис. 6.30.

С помощью команд **GRANT** и **REVOKE** устанавливаем привилегии:

```
GRANT insert, delete ON firm TO Lena
GRANT insert, delete ON country TO Karina
REVOKE insert, delete ON firm FROM Karina
REVOKE insert, delete ON country FROM Lena
GRANT update, select ON country (name_country) TO Karina, Lena
GRANT update, select ON firm (name_firm) TO Karina, Lena
REVOKE update,select ON country (key_country, times_) FROM Karina, Lena
REVOKE update,select ON firm (key_firm, key_country, times_) FROM Karina, Lena
```

Появившиеся права доступа отображаются в Редакторе управления правами доступа, как это показано на рис. 6.31, а более детальную информацию можно получить, как это показано на рис. 6.32.

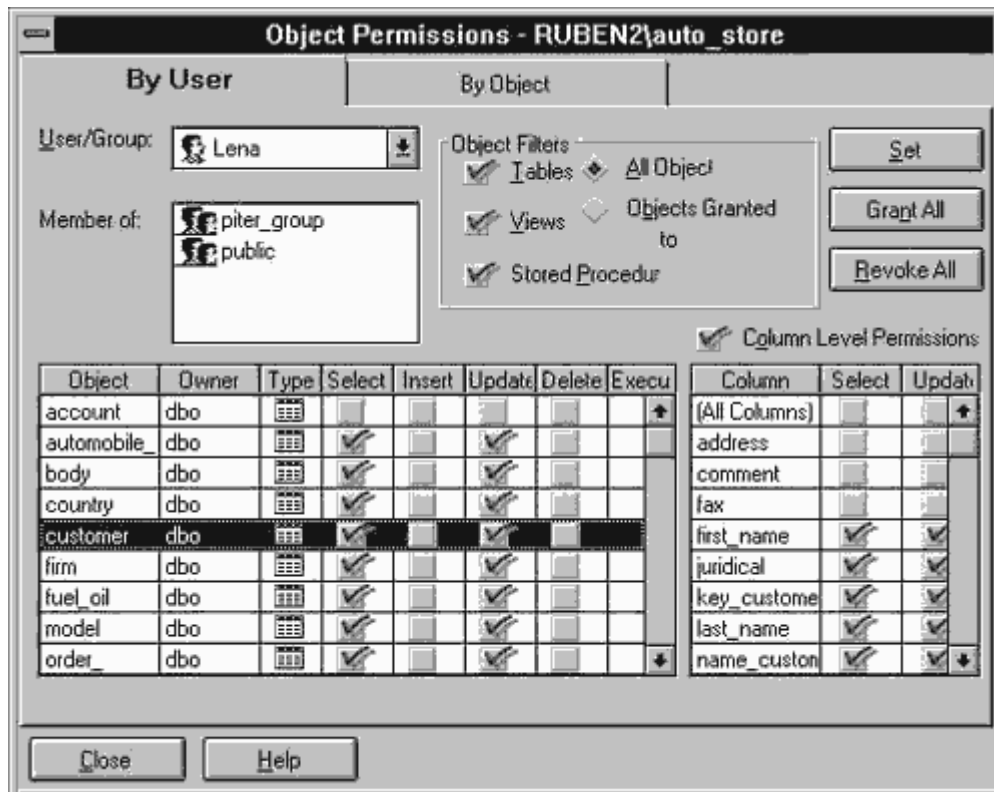


Рис. 6.31.

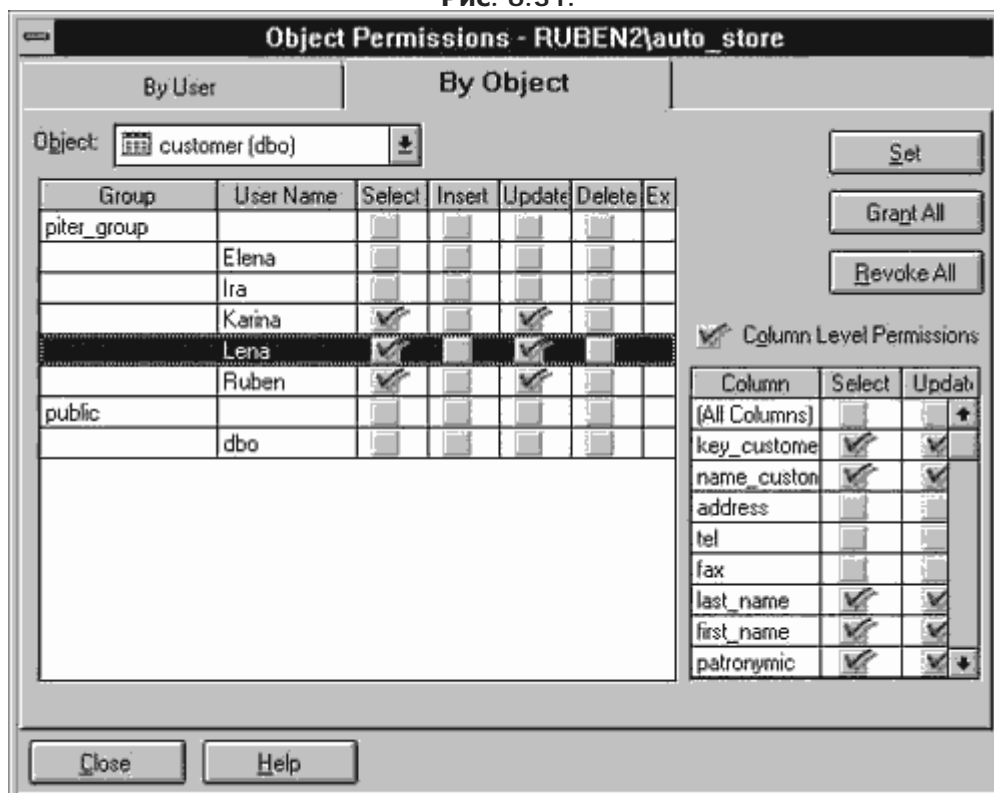


Рис. 6.32.

Планирование процесса наращивания

Наращивание (upsizing) - это процесс, в ходе которого на внешнем сервере создается база данных с той же табличной структурой, с теми же данными и, возможно, со многими другими атрибутами, что и у исходной локальной базы данных, например Visual FoxPro.

С помощью наращивания на базе существующего приложения, которое работает на локальном компьютере или файл-сервере, создается приложение, функционирующее в архитектуре клиент-сервер.

Создавая приложение, которое вы собираетесь наращивать, необходимо так выбирать особенности проектируемой архитектуры приложения и программной модели, чтобы добиться максимальной производительности работы на внешнем источнике данных.

Рассмотрим процесс наращивания приложения, созданного в Visual FoxPro. Этот процесс можно начинать сразу после того, как вы продумаете все детали работы приложения клиент-сервер и будет готов локальный прототип.

Локальный прототип представляет собой рабочую модель приложения, в которой для описания данных используются таблицы и представления Visual FoxPro.

Для переноса баз данных и содержащихся в них таблиц и представлений из локального прототипа на внешний сервер MS SQL Server самый простой путь - использование Мастера наращивания (Upsizing Wizard).

Мастер наращивания создает базу данных SQL Server, которая, насколько это возможно, дублирует функциональные возможности таблиц базы данных Visual FoxPro. Вы также можете переадресовать представления Visual FoxPro, чтобы они использовали не локальные данные, а вновь созданные внешние данные. С помощью Мастера наращивания можно выполнить следующие действия:

- Переместить локальные данные на внешний сервер.
- Преобразовать таблицы локальной базы данных и локальные представления в таблицы внешней базы данных и внешние представления.
- Осуществить миграцию локального приложения в приложение клиент-сервер.

Хотя Мастер наращивания обращается к серверам SQL Server, вы можете создать приложение клиент-сервер для любого внешнего источника данных ODBC. Для серверов, отличных от SQL Server, можно с помощью функций сквозного запроса SQL создать внешние таблицы, а затем с помощью Visual FoxPro создать внешние представления, осуществляющие доступ к таблицам сервера.

Перед проведением наращивания следует убедиться, что вы обладаете необходимыми полномочиями для доступа к серверу, оценить размер базы данных и проверить, достаточно ли места на диске сервера. Кроме того, необходимо предпринять определенные действия в том случае, если наращивание производится на нескольких дисках или устройствах.

В первую очередь убедитесь, что на диске сервера достаточно свободного места. Если Мастер наращивания исчерпает все дисковое пространство на сервере, он прекратит работу, оставив на сервере базу данных и устройства в том объеме, в каком сможет их создать. Вы можете удалить неудачно созданные устройства, базы данных и таблицы с помощью средства администрирования SQL Server.

Чтобы запустить Мастер наращивания, нужно иметь определенные полномочия на работу с сервером SQL Server, на который будет осуществляться наращивание. Диапазон требуемых полномочий зависит от круга решаемых задач:

- Чтобы провести наращивание в существующую базу данных, нужны полномочия CREATE TABLE и CREATE DEFAULT.
- Чтобы построить новую базу данных, нужны полномочия CREATE DATABASE и SELECT на доступ к системным таблицам базы данных Master.
- Чтобы создавать новые устройства, вы должны быть системным администратором.

Когда вы создаете новую базу данных, Мастер наращивания просит выбрать устройства для базы данных и для журнала. Нужно также установить размер базы данных и устройств.

Когда сервер SQL Server создает базу данных, он резервирует фиксированную область пространства для этой базы данных на одном или нескольких устройствах. Не вся эта область обязательно будет использована базой данных. Размер базы данных просто определяет границы, до которых база данных может расти, прежде чем будет исчерпано все место.

После создания базы данных на SQL Server можно увеличить ее размер.

Чтобы оценить размер базы данных, посмотрите, каковы размеры файлов DBF в Visual FoxPro для таблиц, которые вы намерены наращивать, и оцените скорость, с какой будет расти новая база данных на сервере SQL Server. В среднем, каждый мегабайт данных Visual FoxPro требует, по крайней мере, 1,3 - 1,5 мегабайт в среде SQL Server.

Если на диске сервера много места, умножьте размер своих таблиц Visual FoxPro на два. Это гарантирует, что у Мастера наращивания будет достаточно пространства для наращивания базы данных и даже останется место для увеличения объема ваших данных в дальнейшем. Если вы рассчитываете добавлять много информации в базу данных, увеличьте величину этого коэффициента.

Все базы данных и журналы SQL Server размещаются на устройствах. Устройство - это одновременно и логическая область, в которую заносятся базы данных и журналы, и физический файл. Чтобы создать устройство, SQL Server создает файл, тем самым резервируя на диске определенную часть пространства для собственных нужд.

Мастер наращивания показывает, сколько свободного места имеется на существующих устройствах SQL Server. Выберите устройство, на котором объем свободного пространства не меньше, чем оценочный размер базы данных.

Если ни одно из существующих устройств не обладает достаточным количеством свободного пространства, можно создать новое устройство с помощью Мастера наращивания. Новые устройства должны иметь размер, по крайней мере не меньший, чем оценочный размер базы данных. Если это возможно, задайте для устройства больший объем, чем требуется для базы данных, чтобы вы смогли впоследствии расширить ее или поместить на то же самое устройство другие базы данных или журналы.

Размер устройства изменить нельзя. Убедитесь в том, что вы создаете достаточно вместительные устройства.

В большинстве случаев Мастер наращивания обеспечивает контроль над устройствами SQL Server в более чем достаточном объеме. Тем не менее, если на сервере имеется несколько дисков или если вы хотите разместить базу данных или журнал на нескольких устройствах, можно создать устройства до запуска Мастера наращивания.

Если на сервере установлено более одного физического жесткого диска, можно разместить базу данных на одном диске, а журнал базы данных - на другом. При сбое диска у вас будет больше шансов восстановить базу данных в том случае, если она разведена с журналом по разным физическим дискам.

Мастер наращивания позволяет создавать новые устройства, но только на одном физическом диске - на том, который является устройством главной базы данных Master.

Чтобы разместить базу данных и журнал на отдельных дисках, убедитесь, что на обоих дисках имеются достаточно большие устройства, и при необходимости создайте новые устройства. Запустите Мастер наращивания.

SQL Server допускает размещение баз данных и журналов на нескольких устройствах. Однако в Мастере наращивания можно задать только одно устройство для базы данных и одно для журнала.

Чтобы задать несколько устройств для базы данных или журнала, сделайте эти устройства (и только их) устройствами, принимаемыми по умолчанию. Затем запустите Мастер наращивания и выберите Default для устройства базы данных или журнала.

Если для новой базы данных или журнала SQL Server не требуется использовать все устройства, принимаемые по умолчанию, SQL Server задействует только устройства, необходимые для размещения базы данных или журнала.

Перед созданием новой внешней базы данных убедитесь в наличии источника данных ODBC или именованного соединения в базе данных Visual FoxPro, осуществляющей доступ к серверу SQL Server.

Перед проведением наращивания было бы разумно создать резервную копию базы данных (файлов DBC, DCT и DCX). Мастер наращивания не модифицирует файлы DBF, а работает с файлом DBC непосредственно, открывая его время от времени как таблицу, переименовывая таблицы и представляя при создании новых удаленных представлений. Если вы сделали копию базы данных, то сможете вернуть ее в состояние, предшествовавшее началу наращивания, заменив модифицированные файлы DBC, DCT и DCX копиями с исходным содержимым и отменив тем самым все переименования и создание новых представлений.

Мастер наращивания пытается открыть все таблицы в базе данных для монопольного пользования, а если какие-либо таблицы уже открыты и имеют статус совместного пользования, то Мастер наращивания закрывает их и открывает вновь в монопольном режиме. Открытие таблиц перед наращиванием в монопольном режиме позволяет предохранить их от попыток пользователей модифицировать записи. Если какие-либо таблицы нельзя открыть для монопольного использования, Мастер наращивания выдает соответствующее сообщение.

После того как вы создали источник данных ODBC и сделали все необходимые приготовления на компьютере клиента и на сервере, можно приступать к наращиванию.

1. В меню **Tools** выберите команду *Wizards*, а затем выберите *Upsizing*.
2. Последующий процесс выполняется в диалоговом режиме, в привычном стиле Мастеров, используемых в Visual FoxPro. На рис. 6.33 приведен первый шаг работы - выбор локальной БД. Вы можете в любой момент нажать кнопку Cancel и прекратить процесс. На сервере не выполняется никаких действий, пока не нажата кнопка Finish.

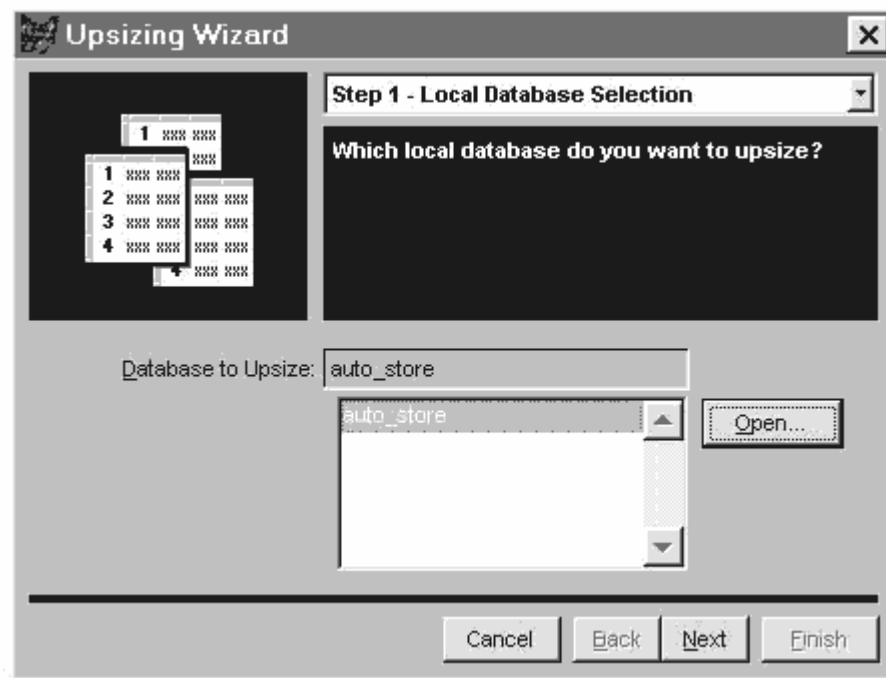


Рис. 6.33. Первый шаг при использовании Мастера наращивания

3. После того как вы нажмете кнопку **Finish**, Мастер наращивания начинает экспортировать базу данных на сервер.

Кнопка **Finish** становится доступной после введения основной информации, требуемой для наращивания. Если нажать кнопку **Finish** до того, как будут пройдены все экраны мастера, Мастер наращивания использует для оставшихся шагов значения, принимаемые по умолчанию.

Наращивание может занять много времени, что зависит от величины данных, объема сетевого трафика и количества запросов, одновременно обрабатываемых сервером. Большие таблицы могут экспортироваться часами. Например, БД, состоящая из трех таблиц, каждая из которых содержала более ста тысяч записей, у авторов переносилась на сервер дольше 8 часов. Мы, конечно, схитрили и запланировали этот грандиозный процесс на ночь, что позволило избежать утомительного ожидания и горячих споров, когда же нажимать клавиши **Ctrl+Alt+Del**. Если вы хотите сократить время переноса БД на сервер, советуем вам при возможности удалить из таблиц поля примечаний и поля типа **General**. В нашем случае после удаления из таблиц полей примечаний процесс переноса занял чуть более 3 часов. После переноса БД эти поля можно восстановить непосредственно на сервере.

Если в процессе экспортирования данных Мастером наращивания возникают какие-либо ошибки, он выдает запрос, нужно ли сохранять информацию об ошибках. Если сохранять информацию нужно - генерируется отчет об ошибках.

Большинство ошибок связано с недостатком места на устройстве базы данных или журнала на используемом сервере или с недостаточным объемом внешней базы данных, которая не в состоянии принять данные, экспортируемые на сервер. Убедитесь, что на выбранных устройствах много свободного места и что размер базы данных установлен достаточно большим.

Чтобы завершить процесс наращивания на сервере, вы можете выполнить следующие действия:

- Убедиться, что таблицы, которые вы намерены редактировать в **Visual FoxPro**, являются обновляемыми.
- Установить полномочия для базы данных, чтобы пользователи могли получать доступ к нужным объектам.
- Защитить результаты работы, сделав новую базу данных восстанавливаемой на случай повреждения или потери данных.

Чтобы таблицу можно было обновлять в **Visual FoxPro**, она должна иметь уникальный индекс. Мастер наращивания может экспортировать имеющийся уникальный индекс, но не в состоянии создать такой индекс, если он отсутствует. Убедитесь, что таблицы, которые вы намерены редактировать в **Visual FoxPro**, являются обновляемыми.

SQL Server назначает своей базе данных и ее объектам набор полномочий, принимаемых по умолчанию. Установите полномочия для внешней базы данных, чтобы пользователи имели доступ

к нужным объектам.

Полномочия, принимаемые для новой базы данных по умолчанию, делают ее доступной только системным администраторам и владельцу базы данных.

Вы можете добавить новых пользователей и группы пользователей, используя на сервере SQL Server приложение Security Manager или системные процедуры `sp_adduser` и `sp_addgroup`.

Подробнее о том, как добавлять пользователей и группы, вы можете узнать из приводимого ниже примера.

Чтобы предоставить полномочия доступа к таблицам, воспользуйтесь приложением SQL Object Manager или командами **GRANT** и **REVOKE**.

Когда на сервере SQL Server создается база данных, в системные таблицы главной базы данных Master добавляются новые записи. Сделав дамп базы данных Master, вы получите резервную копию, отражающую все последние изменения.

Составьте план регулярного сохранения резервных копий содержимого базы данных, чтобы при возникновении серьезной проблемы вы могли восстановить базу данных из резервной копии.

Функция зеркального дублирования устройства выполняет постоянное дублирование информации с одного устройства SQL Server на другое. Если первое устройство даст сбой, то можно перейти на второе, содержащее актуальную копию всех транзакций.

Если ожидаются многочисленные изменения в базе данных в промежутке между выполнением резервных копий и нельзя потерять ни одно из этих изменений, воспользуйтесь средством дублирования устройств. Оно будет наиболее эффективным, когда устройства находятся на отдельных дисках, поскольку если они находятся на одном диске и этот диск даст сбой, будут потеряны оба устройства.

А теперь рассмотрим выполнение процесса наращивания локальной БД с помощью технологии сквозных запросов SQL.

Приложение клиент-сервер может получать доступ к данным на сервере двумя способами:

- С помощью внешних представлений.
- Средствами сквозных запросов SQL.

Внешние представления - это самый популярный и простой метод доступа к внешним данным и их обновления. Мастер наращивания может автоматически создавать внешние представления в базе данных в процессе наращивания, также возможно создание внешних представлений с помощью Visual FoxPro после наращивания. Подробнее о внешних представлениях вы прочитаете в дальнейшем.

Технология сквозных запросов SQL позволяет посылать оператор SQL непосредственно на сервер. Поскольку операторы сквозного запроса SQL выполняются на сервере, они позволяют существенно повысить производительность работы приложения. В табл. 6.7 сравниваются технологии использования внешних представлений и сквозных запросов SQL.

Таблица 6.7. Сравнение внешних представлений и сквозных запросов SQL

Удаленное представление	Сквозные запросы SQL
Базируется на операторе SQL SELECT	Базируется на любом операторе SQL, открывая доступ к операторам определения данных или к выполнению хранимых процедур сервера
Выбирает одно результирующее множество	Выбирает одно или несколько результирующих множеств
Обеспечивает встроенное управление соединениями	Требуется явное управление соединениями
Предоставляет встроенную стандартную информацию обновления для операций обновления, вставки и удаления	Не предоставляет информации обновления по умолчанию
Обеспечивает неявное выполнение SQL и выборку данных	Обеспечивает явное выполнение SQL и управление результатами выборки
Не обеспечивает управление транзакциями	Обеспечивает явное управление транзакциями
Постоянно хранит свойства в базе данных	Обеспечивает временные свойства для курсора сквозного запроса SQL, основанного на свойствах сеанса

Использует асинхронную
постепенную выборку при
выполнении кода SQL

Полностью поддерживает
программируемую асинхронную
выборку

Таким образом, технология сквозных запросов SQL предоставляет следующие преимущества по сравнению с внешними представлениями:

- Вы можете пользоваться функциональными возможностями, специфическими для сервера, например хранимыми процедурами и встроенными функциями на базе сервера.
- Вы можете использовать расширенные возможности SQL, поддерживаемые сервером, а также команды определения данных, администрирования сервера и защиты.
- Вы получаете более полный контроль над операторами **Update**, **Delete** и **Insert** сквозных запросов SQL.
- Вы расширяете возможности контроля над внешними транзакциями. **Visual FoxPro** может обрабатывать сквозные запросы SQL, возвращающие более одного результирующего множества.

Сквозные запросы SQL имеют и ряд недостатков. Основные из них:

- По умолчанию сквозной запрос SQL всегда возвращает не обновляемый моментальный "снимок" внешних данных, хранящихся в активном курсоре представления. Курсор можно сделать обновляемым, установив соответствующие свойства с помощью функции **CURSORSETPROP()**. Напротив, обновляемое внешнее представление обычно не требует установки свойств для обновления внешних данных.
- Команды SQL нужно вводить прямо в окне команд или в программе, не пользуясь графическим конструктором представлений.

Независимо от того, какая технология используется - внешние представления или сквозные запросы SQL, вы можете выполнять запросы и обновлять внешние данные. Практика показывает, что обычно используются как внешние представления, так и сквозные запросы SQL.

Сквозной запрос SQL (**SQL pass-through**) обеспечивает прямой доступ к внешнему серверу с помощью функций сквозного запроса SQL.

Эти функции расширяют возможности доступа к серверу и управления им, предлагаемые представлениями. Вы можете, например, составлять описание данных на внешнем сервере, устанавливать свойства сервера и обращаться к хранимым процедурам сервера.

Сквозной запрос SQL является наилучшим средством для создания результирующих множеств, предназначенных только для чтения, и для использования любого другого синтаксиса SQL. В отличие от представления, которое является результирующим множеством оператора SQL **SELECT**, сквозные запросы SQL позволяют посылать на сервер все что угодно, используя функцию **SQLEXEC()** для системы **Visual FoxPro**. Подробнее об отдельных функциях см. главу 8. В конце данной главы приведен пример использования технологии SQL pass-through на **Visual FoxPro**.

С помощью сквозных запросов SQL вы сами можете создавать курсоры. Механизм сквозных запросов SQL предлагает хотя и более непосредственный, но менее долговечный доступ к серверу, чем представления. Если описания представлений хранятся в базе данных на постоянной основе, то курсоры, созданные сквозными запросами SQL, существуют только в рамках текущего сеанса.

Чтобы с помощью сквозных запросов SQL подсоединиться к внешнему источнику данных ODBC, сначала вызовите функцию **Visual FoxPro SQLCONNECT()**, которая создает соединение. После этого можно пользоваться функциями сквозных запросов SQL **Visual FoxPro** для передачи команд во внешний источник данных на выполнение. Если создать соединение с помощью сквозных запросов SQL, информация об этом соединении не будет сохранена в базе данных в качестве определения именованного соединения.

Как пользоваться функциями сквозных запросов SQL в **Visual FoxPro**?

1. Проверьте способность системы соединить компьютер с источником данных. Воспользуйтесь утилитой **Test ODBC** в составе ODBC или ей подобной.
2. Установите соединение с источником данных с помощью функции **SQLCONNECT()** или **SQLSTRINGCONNECT()**.

Например, если вы соединяете Visual FoxPro с источником данных master на сервере SQL Server, вам следует зарегистрироваться в качестве системного администратора (идентификатор пользователя "sa") с паролем "", выдав следующую команду:

```
lhandle=SQLCONNECT('master','sa','')
```

3. Воспользуйтесь функциями сквозных запросов SQL в составе Visual FoxPro для извлечения данных в курсоры Visual FoxPro и обработки этих данных с помощью стандартных команд и функций Visual FoxPro. Например, можно выдать запрос в таблицу Customer и просмотреть полученный курсор с помощью следующей команды:

```
? SQLEXEC (lhandle, "select * from customer" , ; "my_cursor")
BROWSE
```

4. Отсоединитесь от источника данных с помощью функции **SQLDISCONNECT()**.

```
=SQLDISCONNECT(lhandle)
```

Пример использования технологии SQL pass-through на Visual FoxPro

```
SET CONSOLE OFF
```

```
* Установка соединения с источником данных
```

```
lhandle=SQLCONNECT('master','sa','')
```

```
IF lhandle>>0 && Проверка на наличие связи с SQL Server
```

```
* Следующие три строки программного кода удаляют
```

```
* устройство Rdev_lib и устройство Rdev_log,
```

```
* так как впоследствии мы создаем новые устройства с
```

```
* подобными именами и со своими параметрами
```

```
use_m=SQLEXEC(lhandle,"USE master")
```

```
mlibr=SQLEXEC(lhandle,"sp_dropdevice Rdev_lib, delfile")
```

```
mlogr=SQLEXEC(lhandle,"sp_dropdevice Rdev_log, delfile")
```

```
* Создаем устройство для нашей БД
```

```
mlib=SQLEXEC(lhandle,"DISK INIT ;
```

```
NAME = 'Rdev_lib', ;
```

```
PHYSNAME = 'C:\SQL_60\DATA\SAMP_LIB.DAT', ;
```

```
VDEVNO=1, ;
```

```
SIZE = 5120")
```

```
* Создаем устройство для журнала БД
```

```
mlog=SQLEXEC(lhandle,"DISK INIT ; &
```

```
NAME = 'Rdev_log', ;
```

```
PHYSNAME = 'C:\SQL_60\DATA\SAMP_LOG.DAT', ;
```

```
VDEVNO=2, ;
```

```
SIZE = 2048")
```

```
* Если процесс создания
```

```
* устройств прошел успешно
```

```
* Проверка наличия БД auto_store
```

```
* на сервере, и, если она существует,
```

```
* мы ее удаляем
```

```
IF mlib>>0 AND mlog>>0
```

```
=SQLEXEC(lhandle,"IF EXISTS ( SELECT name FROM sysdatabases ;
WHERE name IN ('auto_store') );
```

```
BEGIN DROP DATABASE auto_store END")
```

```
* Создаем базу данных auto_store, которая будет
```

```
* размещаться на двух
```

```
* устройствах Rdev_lib и устройстве для журнала
```

```
* транзакций базы данных Rdev_log
```

```
creat_data=SQLEXEC(lhandle,"CREATE DATABASE auto_store ;
```

```
ON Rdev_lib = 10 LOG ON Rdev_log = 4")
```

```
* Если процесс создания БД прошел успешно
```

```
IF creat_data>>0
```

```
=SQLEXEC(lhandle,"USE auto_store")
```

```
* Проверка наличия таблицы country на сервере, и, если она
```

```
* существует, мы ее удаляем
```

```
=SQLEXEC(lhandle,"IF EXISTS (SELECT name FROM sysobjects;
```

```

WHERE type = 'U' AND name = 'country');
DROP TABLE country")
* Создаем таблицу Country
cr_country=SQLEXP(lhandle, ;
"CREATE TABLE country (key_country smallint ; IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_country varchar(20) ;
NOT NULL, times_ timestamp)")
* Проверка наличия таблицы Firm на сервере, и, если она
* существует мы ее удаляем
=SQLEXP(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'firm') ;
DROP TABLE firm")
* Создаем таблицу Firm
cr_firm=SQLEXP(lhandle, ;
"CREATE TABLE firm (key_firm smallint IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_firm varchar(20) NOT NULL, ;
key_country smallint REFERENCES country(key_country), ;
times_ timestamp)")
* Проверка наличия таблицы Fuel_oil на сервере,
* и, если она существует, мы ее удаляем
=SQLEXP(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'fuel_oil') ;
DROP TABLE fuel_oil")
* Создаем таблицу fuel_oil
cr_fuel=SQLEXP(lhandle, ;
"CREATE TABLE fuel_oil (key_fuel_oil smallint ; IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_fuel_oil varchar(20) NOT NULL, ;
times_ timestamp)")
* Проверка наличия таблицы Tyre на сервере, и, если она
* существует, мы ее удаляем
=SQLEXP(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'tyre') ;
DROP TABLE tyre")
* Создаем таблицу Tyre
cr_tyre=SQLEXP(lhandle, ;
"CREATE TABLE tyre (key_tyre smallint IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_tyre varchar(20) NOT NULL, ;
times_ timestamp)")
* Проверка наличия таблицы Body на сервере, и, если она
* существует, мы ее удаляем
=SQLEXP(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'body') ;
DROP TABLE body")
* Создаем таблицу body
cr_body=SQLEXP(lhandle, ;
"CREATE TABLE body (key_body smallint IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_body varchar(20) NOT NULL, ;
times_ timestamp)")
* Проверка наличия таблицы Model на сервере, и, если она
* существует, мы ее удаляем
=SQLEXP(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'model') ;
DROP TABLE model")
* Создаем таблицу Model
cr_model1=SQLEXP(lhandle, ;
"CREATE TABLE model (key_model smallint IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_model varchar(20) NOT NULL, ;
key_firm smallint DEFAULT 1 REFERENCES firm(key_firm), ;
swept_volume numeric(5) DEFAULT 1 CHECK ;
(swept_volume>>1) NULL)")
cr_model2=SQLEXP(lhandle, ;
"ALTER TABLE model ADD quantity_drum numeric(2) ;
DEFAULT 1 ;
CHECK (quantity_drum>>=1) NULL, capacity numeric(5,1) NULL, torque numeric(5,1) NULL,
key_fuel_oil ;
smallint DEFAULT 1 ;

```

```

REFERENCES fuel_oil(key_fuel_oil), top_speed numeric(5,1) NULL") cr_model3=SQLEEXEC(lhandle, ;
"ALTER TABLE model ADD starting numeric(4,1) NULL, ;
key_tyre smallint DEFAULT 1 REFERENCES tyre(key_tyre), ;
key_body smallint DEFAULT 1 REFERENCES body(key_body), ;
    quantity_door numeric(1) NULL, ;
    quantity_sead numeric(2) NULL")
cr_model4=SQLEEXEC(lhandle, ;
"ALTER TABLE model ADD length numeric(5) NULL, ;
width numeric(4) NULL, height numeric(4) NULL, ; expense_90 ;
numeric(4,1) NULL, expense_120 numeric(4,1) NULL, ;
expense_town numeric(4,1) NULL, times_ timestamp")
* Проверка наличия таблицы Automobile_passenger_car
* на сервере, и, если она существует, мы ее удаляем
=SQLEEXEC(lhandle,"IF EXISTS (SELECT name ;
    FROM sysobjects ;
WHERE type = 'U' AND name = 'automobile_passenger_car') ;
    DROP TABLE automobile_passenger_car")
* Создаем таблицу Automobile_passenger_car
cr_auto=SQLEEXEC(lhandle, ;
"CREATE TABLE automobile_passenger_car ;
(key_auto smallint IDENTITY(1,1) PRIMARY KEY CLUSTERED, ;
key_model smallint REFERENCES model(key_model), ;
date_issue datetime, cost numeric(10,2), times_ ; timestamp)")
* Проверка наличия таблицы Customer на сервере, и, если
* она существует, мы ее удаляем
=SQLEEXEC(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'customer') ;
    DROP TABLE customer")
* Создаем таблицу customer
cr_customer1=SQLEEXEC(lhandle, ;
"CREATE TABLE customer (key_customer smallint IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, name_customer varchar(30) ;
NOT NULL, address varchar(30), tel varchar(12), ; fax varchar(12), ;
last_name varchar(17), first_name varchar(17))")
cr_customer2=SQLEEXEC(lhandle, ;
"ALTER TABLE customer ADD patronymic varchar(17), juridical bit, comment text, times_ timestamp")
* Проверка наличия таблицы Sale на сервере, и, если она
* существует, мы ее удаляем
=SQLEEXEC(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'sale') ;
    DROP TABLE sale")
&& Создаем таблицу sale
cr_sale=SQLEEXEC(lhandle, ;
"CREATE TABLE sale (account_ smallint ;
PRIMARY KEY CLUSTERED, date_sale datetime, ;
sum_ numeric(10,2), times_ timestamp)")
* Проверка наличия таблицы Account на сервере, и, если она
* существует, мы ее удаляем
=SQLEEXEC(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'account') ;
    DROP TABLE account")
* Создаем таблицу Account
cr_account1=SQLEEXEC(lhandle, ;
"CREATE TABLE account (number_record smallint ; IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, account_ smallint REFERENCES ;
sale(account_), key_customer smallint REFERENCES ;
customer(key_customer), key_auto smallint REFERENCES ;
automobile_passenger_car(key_auto))")
cr_account2=SQLEEXEC(lhandle, ;
"ALTER TABLE account ADD date_write datetime, sold ;
bit, sum_ numeric(10,2), times_ timestamp")
* Проверка наличия таблицы Salesman на сервере, и, если
* она существует, мы ее удаляем
=SQLEEXEC(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'salesman') ;
    DROP TABLE salesman")

```

```

* Создаем таблицу Salesman
cr_salesman=SQLEEXEC(lhandle, ;
"CREATE TABLE salesman (key_salman smallint ; IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, last_name varchar(17), ;
first_name varchar(17), patronymic varchar(17), ; times_ timestamp)")
* Проверка наличия таблицы Order_ на сервере, и, если
* она существует, мы ее удаляем
=SQLEEXEC(lhandle,"IF EXISTS (SELECT name ;
FROM sysobjects WHERE type = 'U' AND name = 'order') ;
DROP TABLE order_")
* Создаем таблицу Order_
cr_order=SQLEEXEC(lhandle, ;
"CREATE TABLE order_ (key_order smallint IDENTITY(1,1) ;
PRIMARY KEY CLUSTERED, key_customer smallint REFERENCES ;
customer(key_customer), key_model smallint REFERENCES ;
model(key_model), key_salman smallint REFERENCES ;
salesman(key_salman), times_ timestamp)")
ELSE
WAIT WIND "Не могу создать базу данных !"
ENDIF
ELSE
WAIT WIND "Не могу создать устройство !" + CHR(13)+ ;
"возможные причины:" +CHR(13)+" - нет места на диске"+CHR(13)+ ;
" - нет полномочий на данную операцию"+CHR(13)+ ;
" - путь указан неверно"
ENDIF
* Регистрация
=SQLEEXEC(lhandle,"USE auto_store")
* Проверка наличия группы Piter_group на сервере, и, если
* она существует, мы ее удаляем
=SQLEEXEC(lhandle,"sp_dropgroup piter_group")
* Создаем группу piter_group
addg=SQLEEXEC(lhandle,"sp_addgroup piter_group")
* Если процесс создания группы прошел успешно
IF addg>>0
* Последовательно создаем процедуры регистрации с
* именами login_lev1, * login_lev2,
* login_lev3, login_lev4, login_lev5 и соответствующими
* им паролями lev1,
* lev2, lev3, lev4, lev5 для БД Auto_store
=SQLEEXEC(lhandle,"sp_addlogin login_lev1, lev1, ; auto_store")
=SQLEEXEC(lhandle,"sp_addlogin login_lev2, lev2, ; auto_store")
=SQLEEXEC(lhandle,"sp_addlogin login_lev3, lev3, ; auto_store")
=SQLEEXEC(lhandle,"sp_addlogin login_lev4, lev4, ; auto_store")
=SQLEEXEC(lhandle,"sp_addlogin login_lev5, lev5, ; auto_store")
* Последовательно создаем пользователей с именами Ruben, Karina, Lena, Elena и Ira для группы
Piter_group
=SQLEEXEC(lhandle,"sp_adduser login_lev1, Ruben, ; piter_group")
=SQLEEXEC(lhandle,"sp_adduser login_lev2, Karina, ; piter_group")
=SQLEEXEC(lhandle,"sp_adduser login_lev3, Lena, ; piter_group")
=SQLEEXEC(lhandle,"sp_adduser login_lev4, Elena, ; piter_group")
=SQLEEXEC(lhandle,"sp_adduser login_lev5, Ira, ; piter_group")
ELSE
WAIT WIND 'Группу PITER_GROUP создать невозможно'
ENDIF
* Заполнение таблиц БД
=SQLEEXEC(lhandle,"USE auto_store")
* Заполнение таблицы Country
=SQLEEXEC(lhandle,"INSERT country (name_country) VALUES ; ('Италия')")
...
=SQLEEXEC(lhandle,"INSERT country (name_country) VALUES ; ('Япония')")
...
* Заполнение таблицы Firm
=SQLEEXEC(lhandle,"INSERT firm (name_firm,key_country) ; VALUES ('Alfa Romeo',1)")
...
&& Заполнение таблицы Fuel_oil

```

```

=SQLEEXEC(lhandle,"INSERT fuel_oil (name_fuel_oil) ;
VALUES ('Дизельное топливо')")
...
&& Заполнение таблицы Tyre
=SQLEEXEC(lhandle,"INSERT tyre (name_tyre) VALUES ('4.5 ; J')")
...
* Заполнение таблицы Body
=SQLEEXEC(lhandle,"INSERT body (name_body) VALUES ; ('Хэтчбек')")
...
* Заполнение таблицы Model
=SQLEEXEC(lhandle,"INSERT model ; (name_model,key_firm,swept_volume, ;
quantity_drum,key_fuel_oil,key_tyre,key_body) VALUES ; ('145 1.4',1,1351,4,3,3,1)")
...
* Заполнение таблицы Automobile_passenger_car
=SQLEEXEC(lhandle,"INSERT automobile_passenger_car ; (key_model,date_issue,cost);
VALUES (1,'7/7/94',10000)")
...
* Заполнение таблицы Customer
=SQLEEXEC(lhandle,"INSERT customer ; (name_customer,address,tel, ;
first_name,juridical) ;
VALUES ('Общество национальных героев','СПб ; ул М Маклая','77710','Юлиан',1)")
...
* Заполнение таблицы Sale
=SQLEEXEC(lhandle,"INSERT sale
(account_,date_sale,sum_) ;
VALUES (101,'5/1/96',10000)")
...
* Заполнение таблицы Account
=SQLEEXEC(lhandle,"INSERT account ; (account_,key_customer,key_auto, ;
date_write,selled,sum_) VALUES ; (101,1,1,'1/1/96',1,10000)")
...
* Заполнение таблицы Salesman
=SQLEEXEC(lhandle,"INSERT salesman ; (last_name,first_name,patronymic) ;
VALUES ('Ажуров','Аристарх','Ариевич')")
...
* Заполнение таблицы Order_
=SQLEEXEC(lhandle,"INSERT order_ ; (key_customer,key_model,key_salman) ;
VALUES (1,1,1)")
...
* Создание привилегий (прав) доступа к полям таблиц БД
=SQLEEXEC(lhandle,"USE auto_store")
=SQLEEXEC(lhandle,"GRANT insert, delete ON model ; TO ruben")
=SQLEEXEC(lhandle,"GRANT update,select ON model ;
TO ruben, karina, lena, elena, ira")
=SQLEEXEC(lhandle,"REVOKE update,select ON customer ; (comment) ;
FROM karina, lena")
...
* Разрываем соединение с источником данных ODBC
=SQLDISCONNECT(lhandle)
ELSE
    WAIT WIND "Связаться с сервером не удастся !"
ENDIF

```

Глава 7

Средства работы с данными

7.1. Организация ввода данных, их поиска и редактирования

Работа с данными в Visual FoxPro

Работа с данными в Microsoft Access

7.2. Создание SQL-запросов

Запросы выборки

- Запросы добавления
- Запросы обновления
- Запросы удаления
- 7.3. Изменение структуры данных с помощью SQL
- 7.4. Запросы и локальные представления в Microsoft Visual FoxPro
- 7.5. Запросы в Microsoft Access
 - Запрос добавления
 - Запрос - Создание таблицы
 - Запрос удаления
 - Запрос обновления
 - Перекрестный запрос
- 7.6. Работа с данными в локальной сети
 - Visual FoxPro
 - Несколько советов по увеличению производительности при работе в сети в приложениях MicroxPro
 - Microsoft Access

Постоянно сталкиваясь с громадным потоком информации, которая обрушивается на нас, не давая передышки ни дома, ни на работе, мы привыкли к невзгодам информационного взрыва. Кажется, что тут сложного, вот она, информация! Поглощай ее глазами с экрана телевизора, лови из радиоприемника, читай, записывай, запоминай. Громадные массивы данных стали заполнять все возможные устройства, которые способны хоть в каком-то виде хранить данные. Почему так быстро растет значение информации? Да потому, что без нее не обойтись не только при управлении государством, маленькой фирмой, без информации трудно надеяться на успех даже не очень дорогой покупки.

Информация нужна всем, но больше всего нужна обработанная информация. Согласитесь, что вам гораздо больше пригодился бы список наиболее низких цен на телевизоры в ближайших к вам магазинах, чем толстый справочник всех магазинов вашего города. Если вы согласны, то перейдем к делу, поговорим об обработке данных.

7.1. Организация ввода данных, их поиска и редактирования

Для работы с данными, хранящимися в таблицах, необходимо организовать ввод данных, поиск и обновление устаревшей информации. Для ввода и редактирования данных лучший способ работы - создание форм. Как это сделать, мы расскажем в [главе 9](#).

В этом параграфе вы узнаете:

- о наиболее эффективных методах поиска данных;
- о способах модернизации данных в отдельных полях и группах записей;
- о том, каким образом в поиске данных могут помочь фильтры;
- и как использовать для работы с данными объекты DAO.

Работа с данными в Visual FoxPro

Данные, для того чтобы отредактировать, необходимо найти. Самый простой способ - вывести данные в табличной форме, например, в окно Browse или воспользоваться объектом Grid. После этого, воспользовавшись клавишами навигации, найти нужную запись и изменить ее. Этот способ вполне пригоден для небольших таблиц, имеющих количество записей, измеряемое двузначным числом. Когда их количество составляет тысячи или даже миллионы, такой поиск затрудняется. В этом случае рекомендуется использовать команды и функции поиска. Одни из них работают без индексов, другие обязательно требуют их создания и подключения.

Одна из наиболее универсальных команд - **LOCATE** - работает без индексов. В то же время наличие индекса по выражению, по которому проводится поиск, без его подключения, значительно повышает скорость поиска. При наличии неподключенного индекса Visual FoxPro использует технологию Rushmore. Эта технология представляет собой специальный метод, существенно ускоряющий процесс поиска нужных данных. В то же время необходимо учитывать, что наличие большого количества индексов требует времени на их обновление.

Для проверки скорости поиска можно использовать следующую процедуру:

- * В таблице Customer существует индекс по полю
- * key_customer.

- * Отключая на всякий случай индексы, мы используем
- * технологию **Rushmore**, что дает нам ускорение поиска
- * более чем в 150 раз при работе с таблицей, имеющей
- * больше 100000 записей.

```
SELECT Customer
SET ORDER TO
nPeriod = SECONDS()
LOCATE FOR key_customer = 45004
nPeriod_1 = SECONDS()
? nPeriod_1 - nPeriod
```

Еще более солидный выигрыш в скорости вы можете получить при работе с символьными полями.

Команда **LOCATE** имеет следующий синтаксис:

```
LOCATE FOR IExpression1
[Scope]
[WHILE IExpression2]
[NOOPTIMIZE]
```

Команда **LOCATE** часто используется в связке с командой **CONTINUE**, которая позволяет продолжить поиск по установленному критерию поиска до тех пор, пока не будут найдены все записи.

Другой способ поиска всех нужных данных - использование фильтров, которые устанавливаются с помощью команды **SET FILTER**. Установка фильтров также дает выигрыш в скорости при наличии индекса по выражению поиска.

Команда **SEEK** позволяет проводить поиск по выражению, для которого имеется индекс и который подключен в момент поиска. Это самый быстрый метод поиска, но требующий постоянного переключения индексов в случае, если необходимо производить поиск по разным выражениям. При использовании этой команды для поиска нужных данных нет необходимости указывать поле, по которому проводится поиск. Таким полем автоматически объявляется активный в данный момент индекс.

```
SELECT Customer
SET ORDER TO key_customer
t=SECONDS()
SEEK 45004
t1=SECONDS()
? t1-t
```

Если вам необходимо найти значение какого-то поля и вам известно значение одного из полей в соответствующей записи, то используйте функцию **LOOKUP()**

```
LOOKUP(ReturnField, eSearchExpression, SearchedField
[, cTagName])
```

Аргументы в этой функции имеют следующее назначение:

- *ReturnField* - поле, значение которого возвратит функция **LOOKUP()**;
- *eSearchExpression* - значение, по которому производится поиск;
- *SearchedField* - поле, в котором производится поиск;
- *cTagName* - имя тега, если есть возможность использовать подключенный индекс.

По скорости поиска функция **LOOKUP()** вполне сравнима с командой **SEEK**. Она также дает выигрыш в скорости при наличии индекса за счет использования технологии **Rushmore**. После успешного поиска указатель записи переводится в ту запись, в которой найдено искомое значение.

Пример использования функции **LOOKUP()**:

```
CLEAR
t=SECONDS()
SET ORDER TO
mlook=LOOKUP(lastname,45004,Key_customer)
```


t1=SECONDS()
 ? t1-t
 ? Mlook

Когда вам необходимо отредактировать одну запись, что, как правило, является обычным условием работы, используйте прямое редактирование полей с помощью форм или окон **Browse**. Если вы хотите, чтобы данные в полях записей редактировались без участия пользователя, то используйте команду **REPLACE**.

Иной раз возникают задачи глобального изменения значений какого-нибудь из полей. К примеру, раньше все были гражданами СССР, а теперь стали гражданами России.

Чтобы одновременно поменять все значения в поле, часто применяется следующая команда:

REPLACE ALL citizenship WITH "Россия"

Тем, кто знаком с SQL, возможно, больше по душе следующая конструкция:

UPDATE kadry SET citizenship = "Россия"

которая работает раза в полтора быстрее.

Тем не менее наибольшей скорости глобального изменения гражданства можно добиться, имея уникальный ключ для всех работников. Например:

```
SELE kadry
SET ORDER TO rabId
  FOR I=1 TO 100000
    SEEK I
    REPLACE citizenship WITH "Россия"
NEXT
```

Данная программа выполнит замену в три раза быстрее, чем программа, использующая команду **REPLACE** с опцией **ALL**. Единственный ее недостаток - это необходимость строгого соблюдения правил присваивания идентификатора каждой записи. В противном случае надо обрабатывать еще и ошибку, которая может возникнуть, если не будет найдена подходящая запись. Если глобальный поиск выполняется часто, можем посоветовать создать индекс по выражению **RECNO()**, то есть по функции, которая возвращает номер записи по порядку.

При обсуждении с заказчиком проекта приложения, структуры базы данных и каждой отдельной таблицы необходимо постоянно думать о предстоящем поиске данных. Учитывая важность этой проблемы при поиске данных, мы сделаем вид, что ничего не писали в первых двух главах и рассмотрим пример влияния структуры таблицы на эффективность поиска данных. Естественно, вы выносите дублирующуюся информацию в отдельные таблицы. Вряд ли имеет смысл хранить технические характеристики автомобиля в таблице счетов. Вы выносите их в отдельную таблицу и при этом каждой записи присваиваете уникальный идентификатор. Если вам надо изменить какую-нибудь из характеристик автомобиля, придется менять данные всего в одной записи одного поля.

Key_auto	key_model	date_ issue	cost	swept_ volume	quantity_ drum	capacity
100001	1	07.07.94	10000	1351	4	90
100002	1	06.06.95	10500	1351	4	90
100003	2	05.10.95	10500	1929	4	90
100004	3	08.09.95	25000	3982	4	286
100005	4	10.12.95	30000	3982	8	286
100006	4	02.01.96	30000	3982	8	286
100007	5	10.10.95	37000	3201	6	321
100008	6	12.06.96	26000	4300	6	193

В приведенной выше таблице сразу видна избыточность данных. Информация в полях **swept_volume**, **quantity_drum**, **capacity** явно дублируется; представьте, сколько записей придется вам редактировать, если характеристики одного из автомобилей надо изменить хотя бы потому, что перед этим они были введены с ошибкой. Разумнее разделить эту таблицу на две, в одной хранить информацию о автомобиле, характерную для каждого отдельного экземпляра, а в другой

- общие характеристики автомобилей. Одна таблица `Automobil_passenger_car` будет выглядеть примерно так:

key_auto	key_model	date_issue	cost
100001	1	07.07.94	10000
100002	1	06.06.95	10500
100003	2	05.10.95	10500
100004	3	08.09.95	25000
100005	4	10.12.95	30000
100006	4	02.01.96	30000
100007	5	10.10.95	37000
100008	6	12.06.96	26000

Вторая таблица - `MODEL` - вот так:

key_model	name_model	key_firm	swept_volume	quantity_drum	capacity
1	145 1.4	1	1351	4	90
2	146 1.9	1	1929	4	90
3	740I 4.0	2	3982	4	286
4	840Ci 4.0	2	3982	8	286
5	M3 3.0	2	3201	6	321
6	GMC Jimmy 4.3	3	4300	6	193

Для связи таблицы используется поле `key_model`. С помощью связующего поля мы легко можем получить любую информацию, которая хранится в этих таблицах.

Вам может понадобиться изменить идентификатор одного из автомобилей в таблице `Model`. Как сделать, чтобы записи не остались одиночными в первой таблице - `Automobil_passenger_car`? Можно провести замену с помощью команды **REPLACE**, то есть найти записи с этим идентификатором и заменить на новое значение. Но лучше использовать ссылочную целостность и написать триггер для выполнения изменений данных в таблице `Model`. В нем следует написать код, который будет искать все соответствующие записи в таблице `Automobil_passenger_car`. Теперь у пользователя будет создаваться впечатление, что умный компьютер сам знает, что нужно поменять идентификатор автомобиля во всех таблицах, в которых он присутствует. Если вам не хочется писать триггеры самостоятельно, то в некоторых случаях `Visual FoxPro` сможет вам помочь, если вы воспользуетесь диалоговым окном `Referential Integrity`, которое можно вызвать из меню *Database*.

Работа с данными в Microsoft Access

Для ввода и редактирования данных в `Microsoft Access` используются формы, которые могут иметь несколько режимов, таблицы и запросы. При открытии формы, по умолчанию, если вы не подключили свое меню, становится активным меню *Форма*, в котором имеются команды для поиска, сортировки и фильтрации данных. Аналогичные команды присутствуют в меню, связанном с таблицами и запросами. Как правило, при простом редактировании данных в таблицах этих средств более чем достаточно.

Среди средств поиска наиболее простым является использование диалогового окна Поиск, с помощью которого вы можете последовательно находить нужные вам записи. Это диалоговое окно позволяет искать данные как в текущем поле, так и в других полях используемого источника данных. Вы можете последовательно находить несколько записей, удовлетворяющих выражению или его части, введенному в строке образца. Поиск можно проводить как вверх по таблице от текущей записи, так и вниз, либо по всей таблице. Но совершенно очевидно, что таким образом поиск можно производить только по одному полю, либо при случайном стечении обстоятельств по нескольким, к примеру, слово "Петрович" может быть и отчеством и фамилией.

В форме вы можете с помощью Мастера кнопок построить кнопку, которая будет служить для поиска записей. После того как вы создадите подобную кнопку с характерным рисунком бинокля или надписью "Поиск записи", откройте окно кода для события `Click`. Скорее всего там будут следующие строки:

```

Sub Кнопка40_Click()
On Error GoTo Err_Кнопка40_Click
    Screen.PreviousControl.SetFocus
    DoCmd.DoMenuItem acFormBar, acEditMenu, 10, , _ acMenuVer70
Exit_Кнопка40_Click:
Exit Sub
Err_Кнопка40_Click:
    MsgBox Err.Description
    Resume Exit_Кнопка40_Click
End Sub

```

По сути, код события Click для этой кнопки просто выполняет команду Найти меню *Правка*. С тем же успехом вы могли создать макрос с единственной макрокомандой Команда меню, имеющей аргументы:

- Строка меню - Форма
- Название меню - Правка
- Команда - Найти

После этого просто перетащите графический образ макроса из вкладки Макрос в любое место формы, и вы получите кнопку, которая будет выполнять то же самое, что и кнопка, созданная Мастером, а если вы назовете макрос "Поиск записи", то и надпись на кнопке будет аналогичной. Ваш макрос делает то же, что и следующая строчка из процедуры, созданной Мастером:

```
DoCmd.DoMenuItem acFormBar, acEditMenu, 10, , acMenuVer70
```

Выполнить этот макрос, когда не активна строка меню Форма, невозможно, впрочем, как невозможно воспользоваться и методом объекта DoCmd DoMenuItem из процедуры Мастера. Мы все время зависим от наличия на экране меню *Форма*. Единственное преимущество кода события по сравнению с нашим макросом заключается в том, что первый обрабатывает ошибку, когда выяснит, что не может выполняться. Будет выдано сообщение, связанное с этой ошибкой, а наш макрос просто "повиснет", выведя на экран окно обработчика ошибок макросов. Впрочем, даже если меню присутствует на экране, то совсем не обязательно, что будет найдена запись, соответствующая введенному образцу.

Если нужная запись не будет найдена, на экран будет выведено стандартное сообщение Access: "Поиск записей в приложении `Microsoft Access' завершен. Элемент не найден"

В качестве примера организации поиска данных рассмотрим создание своей собственной формы, которая будет возвращать то сообщение, которое мы хотим получить в случае неудачного поиска.

Пример формы "ПОИСК" вы найдете в файле базы данных AUTOSTORE.MDB, который находится на дискете, прилагаемой к данной книге. Форма "AUTOMOBIL_PASSENGER_CAR" имеет кнопку для поиска данных в текущем текстовом поле. При ее нажатии вызывается форма "ПОИСК".

В модуле Bookmodule объявляются две переменные: LastForm типа Form и lastcntr типа Control:

```

Dim LastForm As Form
Dim lastcntr As Control

```

Первая служит для передачи переменной формы, а вторая для передачи переменной последнего активного элемента управления для использования их в форме "ПОИСК".

В событии Click кнопки Поиск формы "AUTOMOBIL_PASSENGER_CAR" записываем следующий код:

```

Sub Кнопка40_Click()
    Set LastForm = Screen.ActiveForm
    Set lastcntr = Screen.PreviousControl
    mybook=LastForm.Bookmark
    PoiskSub
End Sub

```

Процедура PoiskSub служит для установки флажка "В текущем поле". Дело в том, что последний элемент управления, который был активным до нажатия на кнопку Поиск, может быть

не связанным ни с каким полем - для него будет отсутствовать понятие "текущее поле". Следовательно, флажок нужно сделать недоступным, так как его значение используется для дальнейшего поиска:

```
Public Sub PoiskSub()
    DoCmd.OpenForm "Поиск"
    Select Case lastcntr.ControlType
        Case 109, 106, 110, 111
            If Not IsNull(lastcntr.ControlSource) Then
                Forms![Поиск].[Flag3] = True
            Else
                Forms![Поиск].[Flag2] = False
                Forms![Поиск].[Flag3] = False
            End If
        Case Else
            Forms![Поиск].[Flag2] = False
            Forms![Поиск].[Flag3] = False
    End Select
End Sub
```

Таким образом мы передали в процедуру обработки события Click последний активный элемент и последнюю активную форму как параметры и при этом выяснили, был ли последний активный элемент управления связанным или нет.

Для поиска мы используем методы из серии Find объекта RecordSet и метод FindRecord объекта DoCmd.

Набор данных для формы мы создаем с помощью свойства RecordSource. Это может быть таблица, запрос или выражение SQL. С помощью свойства RecordsetClone мы получаем ссылку на объект Recordset, указанный в свойстве RecordSource. Если источник данных для формы "AUTOMOBILE PASSENGER CAR" - таблица с тем же именем, то с помощью свойства RecordsetClone формы мы ссылаемся на объект, который создается с помощью следующей конструкции:

```
Dim myDb As DataBase, rst As Recordset
Set Mydb = dbEngine.Workspaces(0).Databases(0)
Set rst = MyDb.OpenRecordset("AUTOMOBIL PASSENGER CAR", _ dbOpenDynaset)
```

Учтите, что объект Recordset функционирует сам по себе, то есть перемещение с помощью методов серии Move и Find не влияет на текущую отображаемую в форме запись. С целью синхронизации перемещения необходимо использовать свойство Bookmark, которое определяет текущую запись. Для этого свойство Bookmark формы сохраняют в переменной строкового типа, а потом присваивают это значение свойству Bookmark объекта Recordset, на который ссылаются с помощью свойства RecordsetClone формы.

В коде кнопки Поиск формы "AUTOMOBIL PASSENGER CAR" имеется следующая строка:

```
mybook=LastForm.Bookmark
```

В коде кнопки Найти для формы "ПОИСК" свойство Bookmark устанавливается следующим образом:

```
LastForm.RecordsetClone.Bookmark=mybook
```

После поиска, если он удачен, значение свойства Bookmark объекта Recordset присваивается свойству Bookmark формы.

Как вы помните, весь этот процесс мы затеяли для того, чтобы выводить свое собственное сообщение при неудачном поиске. При этом мы рассмотрели вопросы взаимодействия формы с объектами доступа к данным (DAO).

Как правило, скорость работы в Access понижается с увеличением числа записей в таблице, что, впрочем, происходит и во многих других приложениях. Для того чтобы снизить негативные последствия этого явления, в Access предлагается использовать развитую систему фильтров. Фильтры можно устанавливать при работе с формой с помощью меню, макрокоманд или методов объекта DoCmd в программном коде.

Например, используя опцию "Фильтр по выделенному", вы можете, последовательно выделяя поля, устанавливать достаточно сложные фильтры. Последовательность действий выглядит следующим образом:

1. Открытие формы.
2. Выделение любого поля или части поля.
3. Выполнение команды *Фильтр по выделенному* из меню *Записи* или нажатие кнопки с тем же названием на панели инструментов Режим Формы.
4. Повторение вышеприведенной последовательности действий с другими полями.

Можно открыть форму в режиме "Форма с установленным фильтром". Это делается с помощью макрокоманды `OpenForm` или с помощью метода `OpenForm` объекта `DoCmd`:

```
DoCmd.OpenForm "automobile passenger car",,,, "key_model=1"
```

Кроме того, вы можете установить источником данных для формы параметрический запрос, который позволяет значительно уменьшить количество записей, обрабатываемых в текущем сеансе работы формы.

При работе с объектами **DAO** можно динамически изменять размер выборки, тем самым значительно ускоряя доступ к нужным вам данным. В качестве примера приведем две процедуры. Одна ищет запись в неотфильтрованном наборе данных, другая использует фильтр.

Процедура поиска в неотфильтрованном наборе данных:

```
Public Sub findexp()
*** В данной процедуре ищется запись, и в окне отладки
*** выводится время поиска в секундах
*** Вы можете подставить свою таблицу
*** и свои поля
*** При этом имеет смысл экспериментировать
*** на таблице с количеством записей более 1000,
*** либо использовать WINAPI для фиксирования времени
*** поиска
Dim myDb As DATABASE
Dim myRst As Recordset
Dim t1 As Double
Dim t2 As Double
Set myDb = DBEngine.Workspaces(0).Databases(0)
*** Можете подставить имя вашей таблицы здесь
Set myRst = myDb.OpenRecordset("first", _ dbOpenDynaset)
t1 = Now()
*** Соответственно поле и критерий поиска
*** тоже необходимо подставить свои
myRst.FindFirst "first like 'Чф*'"
t2 = Now()
If Not myRst.NoMatch Then
Debug.Print myRst!First, myRst!Third
Else
Debug.Print "Пролет"
End If
*** Здесь можете использовать свою
*** функцию для форматирования вывода времени поиска
Debug.Print DateDiff("s", t1, t2)
End Sub
```

Поиск в отфильтрованном наборе данных:

```
Public Sub findexpFilt()
Dim myDb As DATABASE
Dim myRst As Recordset, NmyRst As Recordset
Dim t1 As Double
Dim t2 As Double
Dim i As Long
Set myDb = DBEngine.Workspaces(0).Databases(0)
Set myRst = myDb.OpenRecordset("first", _ dbOpenDynaset)
myRst.Filter = "third >> 56700 and third << 58000"
Set NmyRst = myRst.OpenRecordset()
t1 = Now()
NmyRst.FindFirst "first like 'Чф*'"
```

```

t2 = Now()
If Not NmyRst.NoMatch Then
Debug.Print NmyRst!First, NmyRst!Third
Else
Debug.Print "Пролет"
End If
Debug.Print DateDiff("s", t1, t2)
End Sub

```

7.2. Создание SQL-запросов

Как уже было отмечено, всем, кто хочет работать с базами данных, необходимо знать язык SQL, который, по сути, стал стандартом для работы с базами данных.

В этом параграфе вы узнаете:

- сферы наиболее эффективного применения языка SQL при разработке систем обработки данных;
- основные виды запросов, которые могут быть составлены для работы с данными;
- назначение основных составляющих элементов команд SQL;
- особенности составления команд SQL в рассматриваемых средствах разработки.

К сожалению (а может быть наоборот), рынок программного обеспечения развивается не по принципам единого планирования. Каждый продукт, который использует SQL, применяет его диалект, как правило, отличающийся от ANSI-стандарта этого языка. Обидно, - иначе бы мы могли одинаково обращаться к данным любого продукта, используя одинаковые языковые конструкции. Но в принципе, все еще может быть, а пока приходится довольствоваться тем, что есть. А есть масса приложений для разработки баз данных, которые при этом данные хранят в своем формате, как правило, секрета не представляющем, и все без исключения соревнуются друг с другом в как можно большем количестве отличий от общепринятых стандартов ANSI SQL. Однако общаться программам с данными чужих форматов необходимо, и существует несколько путей для этого общения. Перечислим некоторые из них.

Вы покупаете продукт, который поддерживает несколько форматов. Или - создаете продукт, который поддерживает несколько форматов. Есть прекрасные примеры: Lotus Approach, Microsoft Access, продукты фирмы Borland Dbase и Paradox читают форматы друг друга. Но почти наверняка вы не найдете приложения, которое поддерживает все форматы. А на практике работать с данными иного формата приходится очень часто. Даже если вы очень упорно будете избегать встречи с другими форматами, она все равно когда-нибудь произойдет. Бесспорно, можно воспользоваться следующим способом для работы с внешними форматами, но он таит свои сюрпризы.

Вы используете операции экспорта или импорта. Опять же необязательно, что у вас будут в наличии все необходимые конверторы. Но все же допустим, что они есть. Вы импортировали данные. Отредактировали. Сколько вы поставите на то, что изменения отразились в исходных данных, то есть в файле, где они хранятся в родном формате? Готовы с вами поспорить на любую сумму. Можно, конечно, провести обратную операцию и экспортировать отредактированные данные в исходный формат. Но, согласитесь, что это очень непродуктивный путь для редактирования одной записи в базе данных, где их 100000, а ведь очень часто их бывает много больше. Кроме этого, вам станется недостаточными все триггеры, бизнес-правила и хранимые процедуры для данных внешнего формата, если они, конечно, используются.

Способ, который звучит наиболее современно и который делает вас поистине всесильным, но оставляющий все же место для всевозможных придинок по отношению к себе. Это, само собой разумеется, - OLE 2.0. А в OLE 2.0 нас больше всего интересует его составная часть OLE Automation. Термин, который редко переводится, и мы тоже не возьмемся за это. Придирики здесь могут возникнуть со стороны консервативных любителей DOS. Дело в том, что технологию OLE Automation невозможно реализовать в рамках этой операционной оболочки. Позволим себе лирическое отступление. Вам наверняка придется слышать патетические высказывания, изречаемые достаточно образованными людьми. Они будут гласить, что на их программах работают сотрудники, которым надо только стучать по клавиатуре. Или что наращивание мощности техники для обработки информации - это от лукавого. Не слушайте их. Нам нужны различные, мощные, еще мощнее, самые мощные средства обработки данных. При этом неважно, где вы работаете. Это может быть ЖЭК, банк, спортклуб. Каждый грамм информации имеет право на существование, даже количество бензина, которое вы сегодня израсходовали, отлучившись на служебной машине к вашей любовнице. Не слушайте их, приземленных противников прогресса, которым легче обслуживать свои любительские творения, чем осваивать новые вершины самой нужной профессии современности - специалиста по обработке информации. Бесспорно, на смену

OLE придет нечто более совершенное, но главное, что это не будет шагом назад. OLE Automation - способ управлять и считывать информацию об объектах внешнего по отношению к вашему приложения. Впрочем, разговор об этом чуть ниже.

Следующий способ имеет меньше ограничений. Мы говорим об ODBC (Open Database Connectivity), что можно перевести как Открытый Доступ к Бадам Данных. Конечно, ваше приложение должно поддерживать эту технологию. Установив связь с исходной базой данных, далее вы используете набор SQL pass-through функций для выборки, обработки и модификации исходных данных. Наборы этих функций для разных продуктов неодинаковы, но в целом можно говорить о наборе различных реализаций SQL pass-through функций. Изучив их реализацию в одном языке, вы легко можете воспользоваться аналогичной в другом. Теперь вы очень близки к тому, что во внешней базе данных будут отражаться любые изменения, сделанные в таблицах при редактировании данных. Во всяком случае, необходимый инструмент у вас есть. Настоятельно рекомендуем попробовать. Далее мы обязательно обсудим эту технологию более подробно. Скажем прямо, она-то и является краеугольным камнем всей этой книги. В противном случае это была бы уже совсем другая книга.

Теперь поговорим о реализации SQL языка в каждом приложении. Для общения с данными внешнего формата на его родном языке была изобретена технология ODBC. Впрочем, то же самое мы можем сказать и об OLE 2.0. Запутанно? Осталось только набраться терпения. А дальше мы будем говорить о SQL языке.

В наше время только ленивый не пытается программировать. Тем более, что основные операторы любого из языков достаточно просты, что вполне логично. Быстренько набросали схему данных, загнали все в простую плоскую таблицу, запустили Мастер и пошли искать заказчиков. Дальше может быть хуже, а может быть лучше. Поток информации разрастается, одной таблицы мало (при этом кто-то должен подсказать, что данные из разных таблиц можно связывать), и, когда звучит простой вопрос вашего начальника "А сколько африканского кофе мы продали в Тьмутараканскую область по цене между 6000 и 7000 рублей?", - вы берете листок бумаги, счеты (как наиболее надежное средство учета) и ручку. Правильно ли это? Каждая точка зрения имеет право на существование. В том числе и вашего начальника - в вас начинают сомневаться. Но не пугайтесь, у вас в руках наша книга. Почти 90% дела вы сделали, осталось только проявить чуточку усердия и терпеливо дочитать до конца.

Вспомним, что, несмотря на наличие стандарта SQL, существуют различные реализации или диалекты этого языка, которые являются либо вспомогательными инструментами, либо основой различных систем управления базами данных. Есть и более сложные варианты, когда некоторые СУБД имеют специальные библиотеки для доступа к своим данным для таких языков, как C или Visual Basic с помощью SQL команд приложения. В большинстве случаев к данным любого приложения надо обращаться на его родном SQL диалекте. То есть, находясь в среде FoxPro, вам будет сложно сделать запрос SQL к таблице формата Paradox, не используя ODBC и SQL pass-through. Хотя есть приятные исключения, то есть приложения, которые незаметно для вас проводят всю тяжелую работу по представлению чужих данных в нужном для запроса формате.

В самом простом варианте команда, которая выбирает все записи из одной таблицы, выглядит так:

```
SELECT mytable.* FROM mytable
```

Впрочем, можно еще короче:

```
SELECT * FROM mytable
```

Таким образом, выбраны все записи из таблицы Mytable.

В зависимости от того, какое приложение вы выбрали для изучения SQL, вас ожидают совершенно разные пути ввода данной команды в память вашего компьютера.

Рассмотрим более сложные варианты выбора нужных данных.

Запросы выборки

Начнем с Visual FoxPro. Здесь у нас есть целых три варианта выполнения команд SQL:

1. Набрать команду в окне *Command*.
2. Создать программу, в которую включить нужную команду. Этот способ более эффективен, так как у вас есть возможность сохранить возможно высшей степени изоциренную команду для дальнейшего использования.
3. Способ, наиболее привлекательный для начинающих программистов, - использовать **Relation Query By Example (RQBE) - Реляционный запрос по образцу**.

Для того чтобы использовать RQBE, вам достаточно выполнить команду *New* из меню *File*, а затем выбрать тип создаваемого файла - *Query*. RQBE - это интерактивная среда, в которой вы формируете запрос в основном с помощью мыши, перетаскивая нужные вам поля в определенные области диалогового окна.

Результатом будет запрос, оформленный в виде файла с расширением QPR. При этом выполнить запрос вы сможете с помощью команды **DO**. Если вы изучаете SQL для использования в FoxPro, то пользуйтесь любым из вышеизложенных способов, но если вам нравится RQBE, то все равно заглядывайте в полученный код, потому что это поможет вам лучше понять язык. Разбирайте каждое слово в полученной команде, и успех в недалеком будущем вас будет ожидать всене непременно.

В среде СУБД Microsoft Access нет командной строки в том понятии, в котором она существует в Visual FoxPro. Для того чтобы создать запрос, в контейнере БД выберите страницу Запросы, нажмите на кнопку Создать. На экране появится Конструктор запросов. При этом вы легко будете переходить из Конструктора запросов в окно редактирования SQL или к результатам выполнения запроса.

Среди рассматриваемых средств разработки СУБД Microsoft Access имеет наиболее мощные средства визуального конструирования запросов. Здесь вы можете создать не только запрос на выборку, но и запросы обновления, удаления, создания таблиц, добавления. Об этих запросах мы поговорим ниже.

Visual Basic использует для работы с данными процессор баз данных Microsoft Jet, такой же, как и Access. Но в связи с тем, что Visual Basic является универсальным средством разработки, здесь не присутствуют такие развитые средства визуального проектирования, как в Microsoft Access. В то же время, так как все объекты DAO - объекты по доступу к данным - доступны и из Visual Basic, то мы вполне можем добиться функциональности, которой добиваемся в Access. Другое дело, какой ценой мы этого достигнем.

Напомним, что в Microsoft SQL Server для того, чтобы выполнить команду SQL, необходимо запустить приложение iSQLW. Вы можете набрать требуемую команду и запустить ее на выполнение. При этом на вкладке Query Results вы увидите полученный результат.

Зная SQL, вы можете значительно увеличить ваши возможности при работе с электронными таблицами Microsoft Excel. Несмотря на то, что вы можете непосредственно читать некоторые форматы данных, иногда полезно немного сократить в объеме количество записей перед обработкой в Excel. В этом вам может помочь приложение Microsoft Query, которое можно использовать для построения запросов к данным внешнего и внутреннего формата Excel с целью уменьшения объема данных, необходимых для текущего сеанса работы.

Итак, везде присутствует SQL. Где-то как основное средство работы, где-то как вспомогательное.

Мы думаем, что теперь вы вполне готовы к разговору о более сложных запросах. Посмотрим еще раз на команду SQL, о которой мы уже говорили:

```
SELECT * FROM mytable
```

Заменим абстрактную таблицу Mytable таблицей из примера к этой книге. Одна из таблиц называется Model, в ней хранятся данные о всех моделях автомобилей, которые задействованы в реселлерской деятельности компании. К примеру, нам не нужны для дальнейшей обработки все поля из этой таблицы. Чтобы выбрать нужные, мы должны их просто перечислить:

```
SELECT name_model, swept_volume, quantity_drum FROM model
```

Может случиться, что вам захочется иметь более наглядные заголовки вместо часто абстрактных наименований английскими буквами или даже на английском языке. Тогда мы используем ключевое слово **AS** после названия колонки. Перепишем наш запрос в другом виде:

```
SELECT name_model AS наименование, ;
swept_volume AS рабочий_объем, ;
quantity_drum AS кол_цилиндров ;
FROM model
```

Этот синтаксис верен для Visual FoxPro и Microsoft Access. При работе с Microsoft SQL Server подобная задача выполняется немного по-другому:

```
SELECT 'наименование' = name_model, :
'рабочий_объем' = swept_volume, ;
'кол_цилиндров' = quantity_drum ;
FROM model
```


Итоговый запрос может содержать дублирующие друг друга по всем полям записи. Если вам не нужна избыточная информация, то введите после **SELECT** ключевое слово **DISTINCT**:

```
SELECT DISTINCT 'наименование'= name_model, :
'рабочий_объем'= swept_volume,;
'кол_цилиндров'= quantity_drum ;
FROM model
```

Противоположным по смыслу **DISTINCT** ключевым словом является **ALL**, как правило, оно является значением по умолчанию и его можно не приводить. Но это уже зависит от вашего стиля программирования. Иногда считается правильным указывать все значения и ключевые слова для лучшей "читабельности" кода.

Порядок, в котором поля выводятся в итоге, зависит только от того, в какой последовательности вы перечислите их в своем запросе. Если необходимо во второй колонке вывести количество цилиндров, то следует изменить порядок указания полей в списке выводимых результатов:

```
SELECT name_model, quantity_drum ,swept_volume FROM model
```

Теперь количество цилиндров будет указываться сразу после наименования модели автомобиля.

Помимо полей из таблиц, в список выводимых колонок можно включать функции, переменные, константы и выражения. Например, если вы хотите особо отметить каждую третью запись, вы можете выполнить следующий запрос:

```
SELECT IIF(MOD(RECNO(),3)=0,"Y","N"),RECNO(),cost;
FROM "automobile passenger car"
```

При этом в каждой третьей записи в результирующем курсоре в первой колонке появится буква "Y".

В запросе можно использовать пользовательскую функцию и выводить ее результаты в курсор. Правда, здесь накладывается некоторое ограничение: желательно, чтобы функция не перемещала указатель записи в таблице, из которой выбираются данные.

До сих пор мы рассматривали организацию запросов к одной таблице. В реальной жизни это довольно редкий случай. Как правило, информацию нужно получать из двух и более таблиц. При этом нужно учитывать факт, что таблицы должны быть связаны между собой, при этом необязательно каждая с каждой, можно связать две таблицы через одного или нескольких посредников.

Если вы хотите узнать, в какой стране находится штаб-квартира фирмы, автомобиль которой приобрел каждый покупатель, то нужно создать вот такой запрос:

```
SELECT Customer.name_customer, Country.country_name, ;
Model.name_model,;
Firm.name_firm;
FROM "auto store!customer", "auto store!account",;
"auto store!automobile passenger car" Automobile_passenger_car,;
"auto store!model", "auto store!firm",; "auto store!country";
WHERE Customer.key_customer = Account.key_customer;
AND Automobile_passenger_car.key_auto = Account.key_auto;
AND Model.key_model = ;
Automobile_passenger_car.key_model;
AND Firm.key_firm = Model.key_firm;
AND Country.key_country = Firm.key_country
```

В итоге получится следующая выборка:

NAME _CUSTOMER	COUNTRY	NAME _MODEL	NAME _FIRM
Безумные медведи	Италия	145 1.4	Alfa Romeo
Общество нац. героев	Италия	145 1.4	Alfa Romeo
Пронырливые волки	Италия	146 1.9	Alfa Romeo
Угрюмые слоны	Германия	M3 3.0	BMW

Голодные лисы	Германия 740i 4.0 BMW
Голодные лисы	Германия 840Ci 4.0 BMW
Угрюмые слоны	Германия 840Ci 4.0 BMW

В этом запросе появилось предложение **WHERE**, о котором мы прежде не упоминали. Обратившись к теории, скажем, что это слово позволяет нам определить предикат, который будет обрабатывать каждую строку таблиц, и, в зависимости от его истинности, нужные нам поля из этих строк окажутся в итоговом запросе. В нашем конкретном случае мы просто определяли, есть ли соответствующие значения по связующему полю для таблицы в таблице, с которой мы ее связываем.

Например, строка
Automobile_passenger_car.key_auto = Account.key_auto

ограничит нашу выборку только теми записями, которые имеют одинаковые значения и в таблице **Automobile_passenger_car**, и в таблице **Account** по полю **key_auto**. Проведя соответствующее исследование для других связующих пар, мы поймем, каким образом ограничивается набор выбранных данных. Вышеприведенный синтаксис правилен для любого диалекта SQL, используемого в рассматриваемых средствах разработки.

Рассмотрим более простой пример. Нам хочется узнать для автомобиля название его фирмы-производителя. Запрос будет выглядеть так:

```
SELECT model.name_model, firm.name_firm ;
FROM model, firm ;
WHERE model.key_firm = firm.key_firm
```

Далее рассмотрим, как этот запрос будет выглядеть, если мы построим его с помощью Конструктора запросов Microsoft Access.

```
SELECT DISTINCTROW firm.name_firm, model.name_model
FROM firm INNER JOIN model ON firm.key_firm = model.key_firm;
```

В предложении **FROM** появилась конструкция:

```
firm INNER JOIN model ON firm.key_firm = model.key_firm
```

что в дословном переводе означает: внутренняя связь таблицы **Firm** с таблицей **Model** по полю **key_firm**. Почему нельзя просто все записать в предикат, составленный с помощью предложения **WHERE**? Можно. Но полученный запрос не будет модифицирующим, то есть вы не сможете изменять данные в запросе, а соответственно и редактировать данные в исходной таблице посредством запроса.

Ядро базы данных Microsoft Jet - достаточно гибкое средство в отношении типов и форматов SQL выражений, используемых для создания связей между таблицами. В дополнение к связям, которые вы можете указать в предикате, составляемом с помощью предложения **WHERE**, Microsoft Jet SQL может использовать предложение **JOIN**, чтобы указать внутреннюю, левую внешнюю или правую внешнюю связь, а также неэквивалентные связи, в которых связующим критерием является не равенство значений в связующих полях двух таблиц, а выражение условия, которое должно возвращать истину.

Внутренняя связь может быть сложная, когда вы связываете несколько таблиц, например:

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm ;
```

В вышеприведенном примере мы ищем штаб-квартиру фирмы-производителя каждой модели.

Помимо внутреннего (**INNER JOIN**) объединения, вы можете использовать внешние объединения, при этом различаются левые внешние и правые внешние объединения: **LEFT JOIN** и **RIGHT JOIN**. Используя левое внешнее объединение, вы получаете в итоговом запросе все записи из левой таблицы и только те записи из правой таблицы, которые имеют соответствующие значения по связующему полю в левой таблице.

```
SELECT DISTINCTROW [automobile passenger car].date_issue, account.account
```

```
FROM [automobile passenger car] LEFT JOIN account ON [automobile passenger car].key_auto =
account.key_auto;
```

Какое объединение использовать, зависит только от того, где расположены сравниваемые поля в предложении объединения. Совершенно никакого различия в скорости выполнения или наборе полученных записей вы не получите. Предыдущий запрос мы можем переписать следующим образом без всякой потери производительности:

```
SELECT DISTINCTROW [automobile passenger car].date_issue, account.account
FROM account LEFT JOIN [automobile passenger car] ON account.key_auto = [automobile passenger
car].key_auto;
```

Microsoft Jet возвращает специальное значение **NULL** во всех полях для записей внешней таблицы, не содержащих одинаковых значений. Вы можете использовать это свойство для проверки целостности связи в зависимости от присутствия значения типа **NULL** во внешней таблице. Если в итоговом запросе присутствуют значения типа **NULL**, вы можете быть уверены в наличии несвязанных записей в дочерней таблице, или, наоборот, есть значения в родительской таблице, для которых отсутствуют соответствующие значения в дочерней таблице. Если вы хотите узнать, с каким поставщиком у вас не было сделок, например, за последний месяц, то лучший и самый быстрый способ это сделать - использовать внешнее объединение.

Для сокращения объема выборки, то есть для получения в итоговом курсоре только тех данных, которые вас интересуют в конкретный момент, вы должны использовать предикат, составляемый с помощью предложения **WHERE**, который служит не только для установления связей, но и для наложения фильтров на выбираемые данные.

Допустим, мы хотим выбрать все названия моделей и производящих их фирм в пределах для конкретной страны, например Италии:

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE country.country_name="Италия" ;
```

В SQL вы можете использовать логические операторы **AND**, **OR** и логическое отрицание **NOT**. Добавив всего лишь один оператор **NOT** перед выражением

```
country.country_name = "Италия"
```

мы получаем совершенно противоположную выборку, в которой будут присутствовать фирмы всех стран, кроме Италии.

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE NOT country.country_name="Италия"
```

При необходимости получить данные для двух стран мы используем оператор **OR**. Если нам не нужны данные по автомобилям из Франции и Италии, мы используем следующий запрос:

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE NOT (country.country_name="Италия" OR country.country_name = "Франция")
```

В этом случае необходимо верно расставить скобки, чтобы оператор **NOT** относился и к Италии и к Франции.

Пока мы использовали только один оператор сравнения - равно (=), - на самом деле их несколько больше:

- >> - больше;
- << - меньше;
- >>= - не меньше;
- <<= - не больше;

- <<>> - не равно.

Еще раз вернемся к предыдущему запросу, оставив прежнюю цель - не выбирать модели из Франции и Италии, но при этом используя совсем другие операторы:

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE country.country_name <<>> "Италия" AND country.country_name <<>> "Франция"
```

Обратите внимание, что мы используем **AND** вместо **OR**, иначе в итоговом курсоре были бы выбраны все записи.

Операторы сравнения могут обрабатывать не только числовые значения, но и символьные. При этом обрабатывается ASCII-код символа. При необходимости выполнить запрос, в котором мы хотим выбрать страны, названия которых начинаются на букву "И" и последующие буквы в порядке алфавита, мы можем использовать следующие команды:

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE country.country_name >>="И"
```

В предикате, который формируется с помощью предложения **WHERE**, можно и нужно использовать помимо вышеприведенных еще несколько операторов, а именно: **IN**, **BETWEEN**, **LIKE** и **IS NULL**.

Использование оператора **IN** позволяет нам по-другому выполнить запрос по выборке данных по Италии и Франции. Вспомните запрос, в котором мы собирали данные для всех стран, кроме Италии и Франции. Теперь мы можем записать его с помощью оператора **IN**

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE country.country_name NOT IN ("Италия", "Франция");
```

Оператор **BETWEEN**, строго следуя своему дословному переводу, выводит нам данные в промежутке между значениями, которые мы укажем в качестве его аргументов. Если вам нужно выбрать информацию по странам, которые расположены в алфавитном порядке между Италией и Францией, то используйте запрос, подобный нижеприведенному.

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE country.country_name BETWEEN "Италия" AND "Франция"
```

При использовании этого оператора помните, что краевые значения также попадают в результаты запроса. Если вам это не нужно, то с помощью комбинации предикатов предложения **WHERE** двух последних запросов вы можете добиться необходимого для вашей работы результата.

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE country.country_name NOT IN ("Италия", "Франция") AND (country.country_name BETWEEN
"Италия" AND "Франция")
```

Оператор **LIKE** позволяет использовать в критерии поиска шаблоны. Если необходимо выбрать только модели из стран, названия которых начинаются на букву "И", то можно использовать следующий запрос:

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE country.country_name LIKE "И*"
```

И наконец, оператор **IS NULL** позволяет обнаружить значения типа **NULL** в указанном поле и соответственно определить, выводить проверенную запись в итоговый курсор или нет. Оператор **IS NULL** нельзя непосредственно использовать в **Visual FoxPro** и **Microsoft Access**. Но в этих языках есть функция **ISNULL()**. Поэтому можно написать команду следующим образом:

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE NOT isnull(country.country_name)
```

В случае если вам необходимо отсортировать ваши данные по какому-то полю, вы можете воспользоваться предложением **ORDER BY**. Например, для сортировки данных в предыдущем запросе по названиям фирм воспользуйтесь следующим запросом.

```
SELECT DISTINCTROW firm.name_firm, model.name_model, country.country_name
FROM model INNER JOIN (firm INNER JOIN country ON firm.key_country = country.key_country) ON
model.key_firm = firm.key_firm
WHERE NOT isnull(country.country_name)
ORDER BY firm.name_firm
```

При этом по умолчанию данные будут отсортированы в порядке возрастания. Если необходим противоположный порядок следования записей в итоговом курсоре, то используйте ключевое слово **DESC**, например:

```
ORDER BY firm.name_firm DESC
```

Если бы **SQL**-запросы позволяли нам выполнять только то, что мы успели рассмотреть, то и этого было бы достаточно много, но это еще не все. Мы можем использовать набор агрегатных функций.

Этот набор может различаться в отдельных продуктах. В **ANSI SQL** присутствуют следующие функции агрегирования:

- **COUNT(*)** - возвращает число выбранных записей;
- **COUNT(ALL Expression)** - возвращает количество непустых значений;
- **COUNT(DISTINCT Expression)** - возвращает количество неповторяющихся значений по указанному в *Expression* выражению;
- **MAX(Expression)** - максимальное значение по указанному выражению;
- **MIN(Expression)** - минимальное значение по указанному выражению;
- **SUM(ALL/DISTINCT Expression)** - сумма по всем или только по неповторяющимся значениям;
- **AVG(ALL/DISTINCT Expression)** - среднее по всем или только по неповторяющимся значениям.

В **Microsoft Access** и **Visual FoxPro**, в дополнение к вышеприведенным функциям, вы можете вычислить еще и стандартное отклонение, дисперсию и т. д.

В следующем запросе приводится пример вычисления сумм продаж по конкретным наименованиям автомобилей:

```
SELECT DISTINCTROW Sum(account.sum) AS Sum_sum, model.name_model
FROM model INNER JOIN (account INNER JOIN [automobile passenger car] ON account.key_auto =
[automobile passenger car].key_auto) ON model.key_model = [automobile passenger car].key_model
GROUP BY model.name_model
```

Полученный результат приведен в следующей таблице:

Сумма Наименование	
20500	145 1.4
10500	146 1.9
25000	740i 4.0
60000	840Ci 4.0
37000	M3 3.0

Если внимательно посмотреть на текст запроса, то в конце выражения можно увидеть предложение **GROUP BY**. В стандартном SQL вам просто необходимо группировать запрос по всем выбираемым полям, либо использовать поля как выражения функций агрегирования. Совершенно аналогичные ограничения накладываются на запросы в Microsoft Access. В Visual FoxPro и Microsoft SQL Server таких ограничений нет, и вам не обязательно группировать данные по всем выводимым полям.

При создании запросов перед вами обязательно возникнет необходимость отфильтровать данные по какому-нибудь наименованию. Например, по модели M3 3.0. В принципе, вы можете использовать следующую строку, сформированную с помощью предложения **WHERE**:

```
WHERE model.name_model = "M3 3.0"
```

Но это будет не совсем правильно с точки зрения стандарта SQL. Правильно построенный запрос будет выглядеть следующим образом:

```
SELECT DISTINCTROW Sum(account.sum) AS Сумм, model.name_model
FROM model INNER JOIN (account INNER JOIN [automobile passenger car] ON account.key_auto =
[automobile passenger car].key_auto) ON model.key_model = [automobile passenger car].key_model
GROUP BY model.name_model
HAVING model.name_model = "M3 3.0"
```

Результат этого запроса будет выглядеть следующим образом:

Сумма	Наименование
37000	M3 3.0

Имейте в виду, что критерии, устанавливаемые с помощью **WHERE**, делают выборки, проверяя запись за записью, а предложение **HAVING** отбирает всю группу или агрегат целиком.

Иногда оказывается, что простых запросов недостаточно для получения нужного результата. В таком случае приходится использовать объединение двух запросов или же запросы с подзапросами. Правда, не всегда имеет смысл сразу обращаться к подзапросам, так как они выполняются медленнее. В некоторых случаях лучше приложить усилия для поиска более оптимального решения.

Подзапросы присоединяются к основному запросу через операторы **IN**, **EXIST**, **SOME**, **ANY**, **ALL**.

Рассмотрим пример использования оператора **IN**. Допустим, у нас есть две таблицы с одинаковой структурой. Необходимо вывести данные из первой таблицы при условии, что по полю **kto** у выводимых записей нет совпадающих значений.

```
SELECT kto, skolkо FROM first WHERE kto NOT IN ;
(SELECT DISTINCT kto FROM second)
```

Пример использования оператора **EXIST**. Оператор **EXIST** - единственный из операторов, не требующий выражения между ключевым словом **WHERE** и самим собой. Он возвращает истину в зависимости от того, есть ли хоть одна запись в выборке подзапроса. Рассмотрим два решения одной задачи. Нам необходимо выбрать все модели автомобилей, которые стоят больше 25000 и которые мы ухитрились продать хотя бы один раз. Код записан в синтаксисе FoxPro.

```
SELECT DISTINCT Model.name_model,; Automobile_passenger_car.cost;
FROM "auto store!model",;
"auto store!automobile passenger car" ; Automobile_passenger_car ;
WHERE Model.key_model = ; Automobile_passenger_car.key_model AND ;
Automobile_passenger_car.cost >=25000 AND Exist ;
(SELECT * FROM account,;
"auto store!automobile passenger car" Automobile_passenger_car ;
WHERE account.key_auto = ; Automobile_passenger_car.key_auto)
SELECT DISTINCT Model.name_model, ; Automobile_passenger_car.cost ;
FROM "auto store!model",;
"auto store!automobile passenger car" ; Automobile_passenger_car;
WHERE Model.key_model = Automobile_passenger_car.key_model AND ;
Automobile_passenger_car.cost >=25000 ;
And Automobile_passenger_car.key_auto NOT IN;
(SELECT DISTINCT account.key_auto FROM account)
```

Возможно, вы найдете еще более короткое решение данной задачи, не корите нас, так как мы думаем еще и об учебных целях приводимых примеров. В первом решении с помощью оператора **EXIST** мы просто проверяем таблицу **Account** на наличие записей во внутреннем подзапросе. Во втором решении нас интересует просто список значений, которые у нас имеются, после выполнения внутреннего запроса по полю **key_auto**.

Второе решение дает правильный ответ, а первое неправильный. Почему? Обратимся к теории. Внутренний запрос выполняется только один раз, и внешний запрос при своей работе обращается только к его итогу, раз за разом проходя по всем записям. Естественно, что оператор **EXIST** здесь оказывается совершенно бесполезным. Есть ли выход из положения? Безусловно. Согласно теории, если мы свяжем внешний и внутренний запрос, получив при этом связанный подзапрос, то вынудим внешний запрос обращаться к внутреннему каждый раз во время обработки записей при решении, выводить ли выбранные поля в итоговый курсор. Поэтому мы выводим во внешнем запросе еще одно поле **Automobile_passenger_car.key_auto** и связываем внешний и внутренний запрос по этому полю. Теперь правильный запрос с использованием оператора **EXIST** выглядит так:

```
SELECT Model.name_model, ; Automobile_passenger_car.cost, ;
Automobile_passenger_car.key_auto;
FROM "auto store!model";;
"auto store!automobile passenger car" Automobile_passenger_car;
WHERE Model.key_model = Automobile_passenger_car.key_model AND ;
Automobile_passenger_car.cost > >=25000 And Exist ;
(SELECT * FROM account WHERE ;
Automobile_passenger_car.key_auto=account.key_auto)
```

Не надо после всех вышеприведенных манипуляций пессимистически относиться к оператору **EXIST**. Ведь теперь мы получили внутренний подзапрос в полное управление и можем развить свой запрос еще больше, помня о том, что его результат будет пересматриваться раз за разом после обработки очередной записи во внешнем запросе.

Пример использования операторов **ANY** и **SOME**. Использование операторов **ANY** и **SOME** приводит к совершенно одинаковым результатам. В качестве примера решим предыдущую задачу:

```
SELECT Model.name_model, Automobile_passenger_car.cost;
FROM "auto store!model";;
"auto store!automobile passenger car" Automobile_passenger_car;
WHERE Model.key_model = Automobile_passenger_car.key_model AND ;
Automobile_passenger_car.cost > >=25000 And Automobile_passenger_car.key_auto= ;
ANY (SELECT DISTINCT account.key_auto FROM account);
```

Пример использования операторов **ALL**. Оператор **ALL** используется с операторами сравнения **>>** и **<<** и подзапросом. Оператор **ALL** с оператором **>>** перед ним означает, что сравниваемое значение должно быть больше любого значения в списке значений, полученных в подзапросе. В противоположном случае - соответственно меньше любого значения в списке.

```
SELECT Model.name_model, Automobile_passenger_car.cost;
FROM "auto store!model";;
"auto store!automobile passenger car" Automobile_passenger_car ;
WHERE Model.key_model = Automobile_passenger_car.key_model and ;
Automobile_passenger_car.cost > >=ALL ;
(SELECT DISTINCT summa FROM account ;
WHERE summa>>=30000)
```

Запросы добавления

Не стоит особо пропагандировать необходимость добавления записей в таблицы в системах обработки данных. Они ведь и становятся настоящими таблицами, когда в них появляются записи. За этот процесс в языке **SQL** отвечает команда **INSERT**. Она имеет два варианта использования. В первом случае вы добавляете одну запись с конкретными данными в конкретные поля. Во втором случае вы можете добавить одну и более записей, набор которых формируется запросом. Необходимо отметить, что второй вариант синтаксиса не реализован в версии **Visual FoxPro 3.0**. Тем не менее его легко реализовать в два шага. Рассмотрим конкретные примеры.

В таблицу **Account** добавим одну запись:

```
INSERT INTO account VALUES (106,6,100007,{12.07.96},.T.,26000)
```

При этом мы не указываем, в какие поля мы добавляем значения. Следовательно, SQL считает, что мы будем добавлять значения во все поля. Далее, после ключевого слова **VALUES** мы в скобках просто перечисляем набор значений, при этом их порядок должен соответствовать порядку полей в таблице и, естественно, совпадать с ними по типу данных.

В случае, если мы заполняем не все поля, нам необходимо явно перечислить те поля, в которые мы будем добавлять значения.

```
INSERT INTO Account (account,key_customer,key_auto,date_write)
VALUES (106,6,100007,{12.07.96})
```

Какой информацией заполнятся поля для добавленной записи, очень сильно зависит от продукта, в котором выполняется операция, и от того, как спроектирована таблица. Это может быть значение по умолчанию для данного поля, значение типа **NULL** или пустая строка для символьных полей и 0 для числовых. Поля можно, как правило, перечислять в произвольном порядке, при этом значения тоже должны соответствовать порядку следования полей в перечислении.

Следующий вариант использования команды **INSERT** ведет к добавлению в таблицу одной и более записей. В нашем случае мы просто добавляем записи из одной таблицы в другую, при этом обе таблицы имеют одинаковую структуру.

```
INSERT INTO mytable ( kto, skolk )
SELECT DISTINCTROW acmytable.kto, acmytable.kto
FROM Acmytable
WHERE (((acmytable.kto)="cd"))
```

Совершенно естественно, что соответствующие поля в выборке заполнения и в выборке добавления должны быть одного типа и размерности.

Запросы обновления

Запросы обновления, которые начинаются с ключевого слова **UPDATE**, служат для замены значений в полях таблицы в записях, которые выбираются по определенному критерию. Если вы не зададите никакого критерия, то обновляются все записи по указанному полю. Вы можете обновлять одно или несколько полей, но только одной таблицы. Есть некоторые различия в синтаксисе этой команды для приложений, которые мы рассматриваем. Самый описательный синтаксис в Microsoft SQL Server

```
UPDATE [[database.]owner.]{ table_name | view_name}
SET [[[(database.)owner.]{ table_name. | view_name.}]
column_name1 = {expression1 | NULL | (select_statement)}
[, column_name2 = {expression2 | NULL | (select_statement)}...]
[FROM [[(database.)owner.]{ table_name | view_name}
[, [(database.)owner.]{ table_name | view_name}]]...]
[WHERE search_conditions]
```

Из приведенного синтаксиса видно, что вы можете связывать изменения в таблице в зависимости от связи с конкретной таблицей. Притом указывать это в предикате, связанном с предложением **FROM**. Таким образом, мы можем сформировать следующую строку

```
UPDATE Account SET summa=summa*1.25
FROM Account, sale
WHERE Account.account=Sale.account AND Sale.date_sale>>{10.01.96}
```

В Microsoft Access, несмотря на отсутствие в синтаксисе предложения **FROM**, мы тем не менее можем легко связывать две таблицы в запросе и обновлять данные в зависимости от значения в другой таблице.

```
UPDATE DISTINCTROW account
INNER JOIN sale ON account.account = sale.account
SET account.summa = summa*1.25
```



```
WHERE (((sale.summa)>>20000));
```

В Visual FoxPro мы можем связать таблицы в предложении **WHERE** и в принципе добиться того же результата.

```
UPDATE Account ;
SET Summa=summa*1.25
WHERE Account.account=sale.account AND Account.date_sale
```

Запросы удаления

Запросы удаления служат для удаления строк из таблицы. Синтаксис для всех рассматриваемых нами диалектов сходен. Вам нужно указать таблицу, из которой вы собираетесь исключить строки. Пример удаления строк

```
DELETE FROM Account WHERE Account.key_auto = 100008
```

Во всех командах, управляющих содержимым данных таблиц, можно применять подзапросы.

7.3. Изменение структуры данных с помощью SQL

Язык SQL может помочь программисту не только более эффективно организовать получение пользователем приложения нужных данных, но и программно создавать и изменять структуру данных.

В этом параграфе вы научитесь программным путем:

- создавать таблицы и индексы;
- изменять структуру существующих таблиц;
- создавать связи между таблицами.

Data Definition Language (DDL) - это раздел языка SQL, который служит для создания таблиц, изменения их структуры или их удаления. Сюда же входит создание индексов. Несмотря на то, что практически во всех настольных СУБД имеются собственные средства создания таблиц, как правило, вы можете создавать таблицы и с помощью языка SQL. При этом за внешним визуальным прикрытием скрываются те же SQL команды. Поэтому логичней использовать возможность докопаться до сути происходящих процессов.

Следующим аргументом для изучения этого раздела SQL может служить задача решения проблемы перевода своей информации с технологии одиноко стоящих компьютеров, обменивающихся информацией посредством дискет, либо технологии файл-серверов, на технологию клиент-сервер.

Рассмотрим в качестве невероятного примера такую простую задачу, как переход вашей организации с платформы Visual FoxPro на Microsoft Access. Несмотря на всю кажущуюся легкость этого процесса, вы понесете немало потерь. Ни одно из свойств таблицы - заголовок поля, значение по умолчанию, правила проверки ввода значений уровня поля и уровня записи - не переносится. Вам все придется создавать заново. В то же время команды **SQL Data Definition Language** могут обеспечить управление импортом данных в Access или экспортом их из FoxPro. Как правило, любое мало-мальски приличное современное средство управления базами данных имеет набор функций **SQL pass-through**, которые позволяют управлять сторонними приложениями с помощью диалекта языка SQL самого стороннего приложения.

Команда **CREATE TABLE** служит для создания таблиц. Ее полный синтаксис выглядит так:

1. Visual FoxPro

```
CREATE TABLE | DBF TableName1 [NAME LongTableName] [FREE]
(FieldName1 FieldType [(nFieldWidth [, nPrecision]))]
[NULL | NOT NULL]
[CHECK IExpression1 [ERROR cMessageText1]]
[DEFAULT eExpression1]
[PRIMARY KEY | UNIQUE]
[REFERENCES TableName2 [TAG TagName1]]
[NOCPTRANS]
[, FieldName2 ...]
```

```
[, PRIMARY KEY eExpression2 TAG TagName2
|, UNIQUE eExpression3 TAG TagName3]
[, FOREIGN KEY eExpression4 TAG TagName4 [NODUP]
REFERENCES TableName3 [TAG TagName5]]
[, CHECK lExpression2 [ERROR cMessageText2]]]
| FROM ARRAY ArrayName
```

2. MS Access

```
CREATE TABLE таблица
(поле1 тип [(размер)] [индекс1] [, поле2 тип [(размер)]
[индекс2] [, ...]] [, составной_индекс [, ...]])
```

3. MS SQL Server

```
CREATE TABLE [(database.)owner.]table_name
(column_name datatype [NOT NULL | NULL]
[, column_name datatype [NOT NULL | NULL]]...)
[ON segment_name ]
```

В зависимости от приложения синтаксис может различаться, но те элементы, которые мы можем создавать с помощью вышеприведенного синтаксиса, можно создавать с помощью других команд. То есть команда **CREATE TABLE** как бы размыкается по нескольким командам, что в значительной мере зависит от внутренней архитектуры используемого процессора баз данных. Синтаксис SQL в Microsoft Access ближе к Microsoft SQL Server, в то время как Visual FoxPro стоит несколько особняком. Хотя в конечном итоге все элементы, которые мы можем создать для таблиц в Access, содержит и MS SQL Server.

Иногда перед вами может стать задача изменения структуры таблицы. Для этого используется команда **ALTER TABLE**. Естественно, что никто не лишает вас возможности сделать это визуальными средствами СУБД, в которой вы проектируете свое прикладное приложение. Очень сложно придумать ситуацию, когда вам необходимо использовать команду **ALTER TABLE** внутри СУБД, где вы проектируете приложение, но, возможно, вам придется менять структуру таблицы, для которой на компьютере нет создающего ее приложения. Но есть драйвер ODBC (иначе ситуация фатальная). Или другой случай, когда вам необходимо с рабочей станции, например, из среды приложения, написанного на Visual FoxPro, добавить поле (колонку) в таблицу, хранящуюся на MS SQL Server.

Самый насыщенный ключевыми словами синтаксис этой команды в Visual FoxPro:

```
ALTER TABLE TableName1
ADD | ALTER [COLUMN] FieldName1
FieldType [(nFieldWidth [, nPrecision])]
[NULL | NOT NULL]
[CHECK lExpression1 [ERROR cMessageText1]]
[DEFAULT eExpression1]
[PRIMARY KEY | UNIQUE]
[REFERENCES TableName2 [TAG TagName1]]
[NOCPTRANS]
Или
ALTER TABLE TableName1
ALTER [COLUMN] FieldName2
[NULL | NOT NULL]
[SET DEFAULT eExpression2]
[SET CHECK lExpression2 [ERROR cMessageText2]]
[DROP DEFAULT]
[DROP CHECK]
Или
ALTER TABLE TableName1
[DROP [COLUMN] FieldName3]
[SET CHECK lExpression3 [ERROR cMessageText3]]
[DROP CHECK]
[ADD PRIMARY KEY eExpression3 TAG TagName2]
[DROP PRIMARY KEY]
[ADD UNIQUE eExpression4 [TAG TagName3]]
[DROP UNIQUE TAG TagName4]
```

```
[ADD FOREIGN KEY [eExpression5] TAG TagName4
REFERENCES TableName2 [TAG TagName5]]
[DROP FOREIGN KEY TAG TagName6 [SAVE]]
[RENAME COLUMN FieldName4 TO FieldName5]
[NOVALIDATE]
```

Намного проще эта команда выглядит в Microsoft Access:

```
ALTER TABLE таблица {ADD {COLUMN поле тип[(размер)] [CONSTRAINT индекс] I
CONSTRAINT составной_индекс} |
DROP {COLUMN поле I CONSTRAINT имя_индекса} }
```

И наконец, синтаксис этой команды в MS SQL Server еще короче:

```
ALTER TABLE [[database.]owner.]table_name ADD column_name datatype NULL [, column_name
datatype NULL...]
```

На этом мы закончили обзор команд SQL, которые присутствуют во всех трех приложениях.

Не удивляйтесь, что параграф уже закончился, а примеров все еще нет. Примеры по использованию команд SQL для изменения структуры данных вы уже видели в [главе 6](#) при описании создания БД.

7.4. Запросы и локальные представления в Microsoft Visual FoxPro

В этом параграфе мы подробно рассмотрим визуальные и программные методы создания запросов и представлений в Visual FoxPro.

Расширения команд xBase командами SQL, которые появились, начиная с версии FoxPro 2.0, принесли, несомненно, ему очень большую пользу. Главным достоинством можно считать значительное сокращение кода для выборки данных, причем без потери скорости, так как метод оптимизации **Rushmore** работает и в SQL запросах. Это единственный вариант, где указанный метод может использоваться для поиска данных в нескольких таблицах. А, например, команда **INSERT-SQL** работает на порядок быстрее, чем классическая конструкция

APPEND BLANK

```
REPLACE <<поле>> WITH <<выражение>>
```

Правда, SQL команд и было всего ничего:

- **SELECT-SQL**
- **INSERT-SQL**
- **CREATE TABLE**

При этом курсоры, которые получались в результате выборки, были немодифицирующими, то есть изменения, которые вы делали в них, не отражались в таблицах, данные из которых они отражали. Тем не менее были изобретены сотни способов синхронизации изменений в курсорах, полученных с помощью команды **SELECT** с данными в исходных таблицах. Одни из самых простых - использование в запросе в качестве выводимых колонок функции **RECNO()**, которая возвращает номер записи и использование возможности **SELECT-SQL** создавать таблицы. У этого способа есть существенное ограничение - он работает корректно только при выборке из одной таблицы. В случае выборки из нескольких таблиц получить с помощью функции **RECNO()** нужный результат невозможно.

Большое число задач требует просто просмотра или распечатки промежуточных результатов, вследствие чего язык SQL прижился и был быстро освоен программистами.

В Visual FoxPro добавилось несколько новых команд SQL, с помощью которых мы можем менять структуру уже существующих таблиц, удалять записи и модифицировать данные. Но самое главное, - у нас теперь есть возможность создавать представления (View), которые можно хранить в базе данных. Тем самым значительно увеличились возможности организации данных. Помимо этого представление позволяет создавать курсоры, с помощью которых мы можем менять данные в исходных таблицах. При этом мы сами можем определять, какие поля разрешить для модификации, а какие нет. Возникают определенные трудности, если в исходных таблицах нет уникальных ключевых полей, но они легко преодолимы. Можно сказать, что запросы в том виде, как они существовали в предыдущих версиях и продолжают существовать сейчас, больше не нужны. Хотя бесспорно при желании можно доказать необходимость их использования. В Visual FoxPro как синтаксис команд по созданию представлений и запросов, так и конструкторы для создания этих краеугольных камней систем управления базами данных очень похожи друг на друга. Поэтому, начиная обзор Конструктора представлений, мы одновременно изучаем и Конструктор запросов. Параллельно мы будем обсуждать как визуальный способ создания

представлений, так и программный.

Вызвать на экран Конструктор представлений можно несколькими способами. Если вы работаете с использованием *Project Manager*, а это единственно правильный путь для построения сложных приложений, то вам необходимо перевести курсор на пункт списка *Local Views* и нажать кнопку *New*. После чего на экране появится диалоговое окно, в котором вам предложат создать представление с помощью Мастера или самостоятельно. Самостоятельно - значит с помощью Конструктора представлений, что ненамного сложнее. Помимо этого вы можете щелкнуть по значку *New* в стандартной панели инструментов, и в появившемся диалоге среди предложенных типов файлов выбрать *Local View*.

Перед тем как на экране появится Конструктор представлений, на экран будет выведено диалоговое окно для выбора таблиц или предварительно созданных представлений, в котором вы должны выбрать соответственно таблицу или представление, на основе данных из которого вы будете создавать новое представление. Если вам необходимо выбирать данные из нескольких таблиц или представлений, то, нажав правую кнопку мыши, находясь внутри Конструктора представлений, вы можете вызвать всплывающее меню и выбрать в нем команду *Add Table*, как это показано на рис. 7.1.

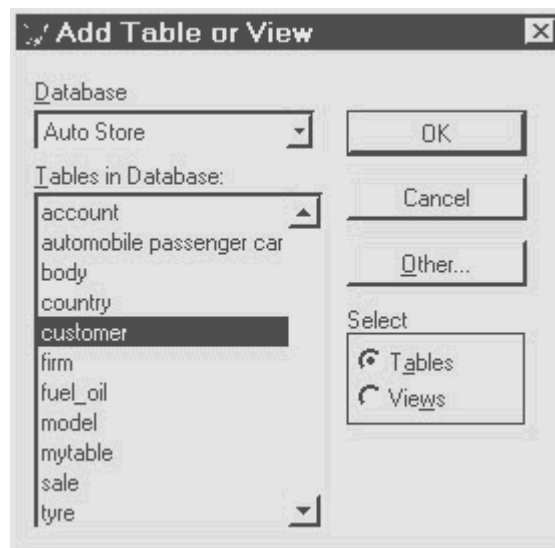


Рис. 7.1. Диалоговое окно выбора таблиц или представлений

Конструктор представления состоит из двух частей. В первой, верхней части, размещаются графические образы таблицы и представления, данные из которых вы будете использовать для создания представления. Здесь легко можно связать две таблицы или представления, просто щелкнув мышью на названии поля и, не отпуская кнопки, переместив курсор на поле в другой таблице или представлении (рис. 7.2). Это очень важный момент, потому что, как правило, все таблицы и представления должны быть связаны, иначе результаты выборки трудно прогнозировать.

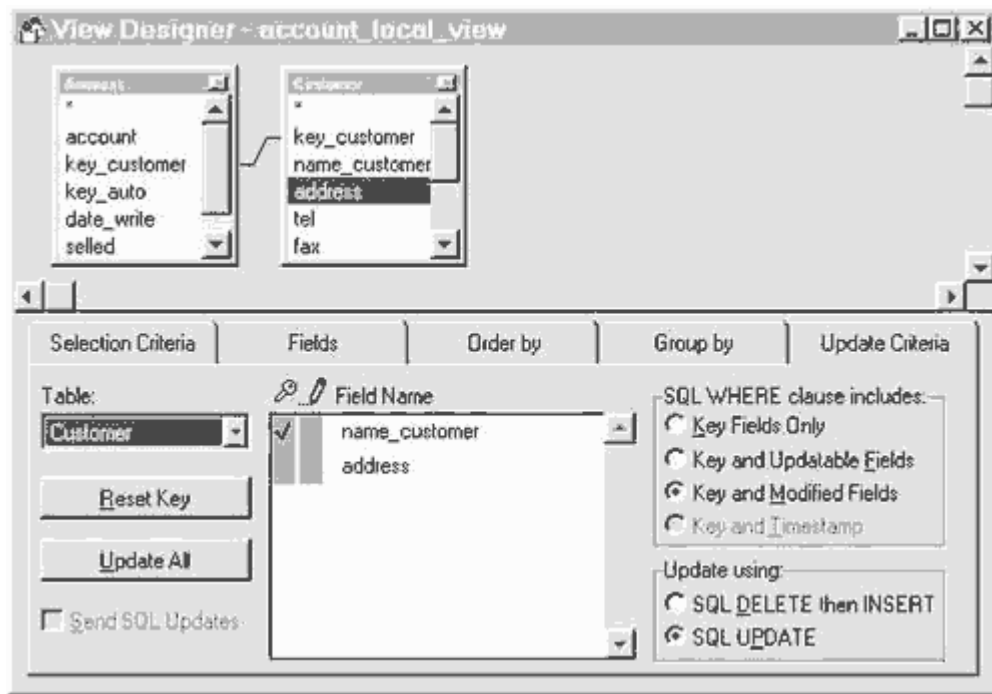


Рис. 7.2.

Нижняя часть Конструктора состоит из пяти страниц, заголовки которых называются Selection Criteria, Fields, Order By, Group By и Update Criteria (рис. 7.3). Здесь стоит отметить, что последней страницы нет в Конструкторе запросов. Это единственное, но очень принципиальное различие между запросом и представлением - представления могут изменять исходные таблицы при изменении данных в них.

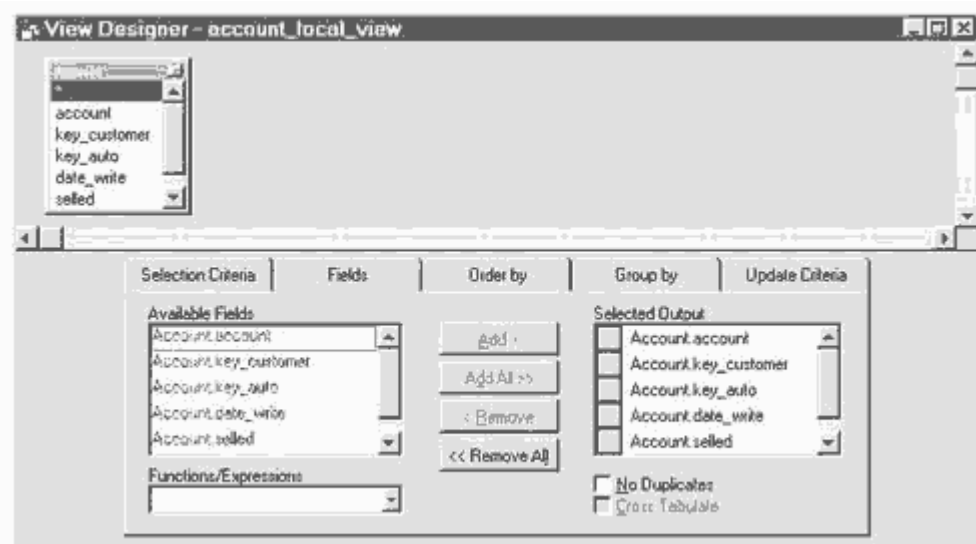


Рис. 7.3.

На странице Selection Criteria записываются связи между таблицами и накладываются фильтры на выбираемые данные. Здесь мы фактически записываем условия предложения WHERE. Если вы еще слабо знакомы с синтаксисом SQL, то есть возможность изучать его в интерактивном режиме. В любой момент вы можете либо с помощью вызова всплывающего меню, или с помощью меню *Query*, или с помощью панели инструментов View Designer просмотреть SQL запрос в текстовом виде. При этом вы увидите, что как и в случае с запросами, так и с представлениями, выражение начинается с ключевого слова SELECT (рис. 7.4). Но пусть вас это не смущает, команда, создающая представления программным путем, начинается все-таки с ключевого слова **CREATE SQL VIEW**. Но об этом разговор чуть ниже.

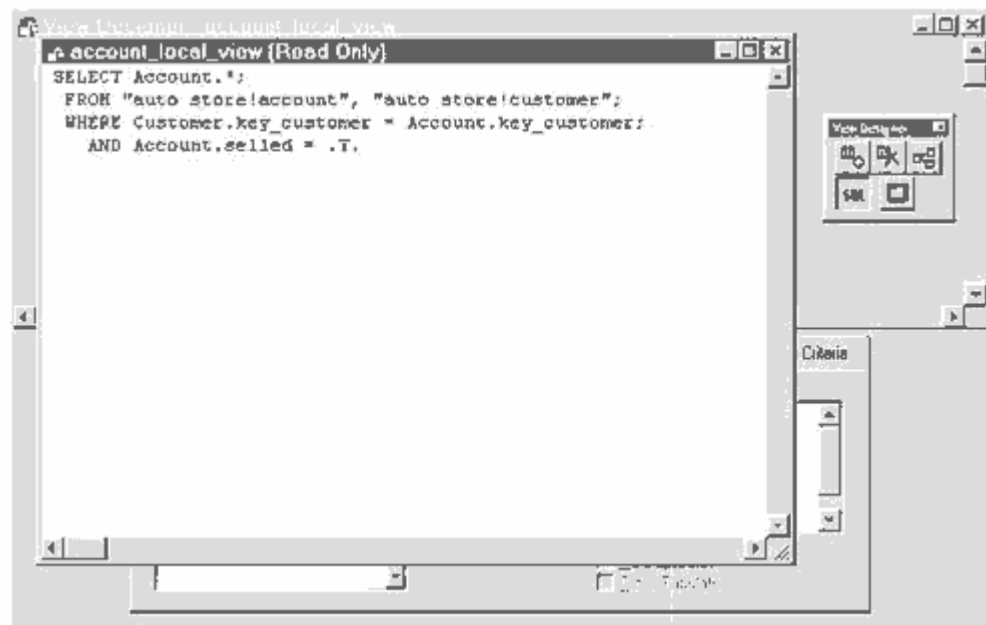


Рис. 7.4. Генерируемый в представлении код SQL запроса

Страница Fields служит для выбора полей из представлений и таблиц, участвующих в создании нашего представления. Можно выбрать поля другими способами, например, дважды щелкнув мышью на названии нужного вам поля или, что более эффектно, просто перетащив графический образ этого поля с образа таблицы в список **Selected Output**, который присутствует на страницах **Fields** и **Selection Criteria**. У визуального способа есть одно ограничение. В случае, если вам необходимо иметь в выборке не данные из поля, а, например, выражение, то вам придется воспользоваться комбинированным списком **Functions/Expressions**. К этому же списку вам придется обратиться в случае, если вы захотите дать колонкам свои имена с помощью опции **AS**.

Если вы внимательно читали параграф о SQL командах, то помните, что в команде **SELECT-SQL** есть предложение **ORDER BY**, которое позволяет вам упорядочить данные по определенным полям. Для этого служит страница **Order by** в Конструкторах представлений и запросов.

В случае использования функций агрегирования вам придется обратиться к странице **Group by** для группировки данных.

Последняя страница называется **Update Criteria**. Именно на этой странице мы можем выбрать поля, редактирование которых в курсоре представления будет отображаться в исходных таблицах. На этой странице в списке **Field Name** отображаются поля из таблиц, к которым обращается запрос представления. При этом отображать можно как одновременно все поля из всех таблиц, так и только из одной таблицы, выбрав соответствующее значение в раскрывающемся списке **Table** (см. рис. 7.2).

Далее для каждой таблицы необходимо выбрать ключевые поля. Обратите внимание, что рядом со значком, изображающим ключ над списком **Field Name**, появится значок, изображающий карандаш, и вертикальная полоска ниже этого изображения. Теперь мы можем пометить поля как модифицируемые. Ключевое поле может быть одновременно и модифицируемым. Если есть необходимость отключить ключевые поля, то нажмите кнопку **Reset**. Когда вы хотите сделать сразу все поля модифицируемыми, нажмите кнопку **Update All**. При этом поля, которые вы укажете как ключевые, помечены не будут. Затем необходимо включить переключатель **SendUpdates**, так как только после этого исходные таблицы будут модифицироваться. Теперь с помощью кнопок выбора **SQL WHERE clause includes** необходимо выбрать одно из четырех значений **WhereType**. Оно будет определять, каким образом станет происходить поиск записи в исходной таблице:

- При выборе значения **Key Fields only** данные будут обновляться в соответствии с изменением значений ключевых полей. Поэтому не забудьте, что если в выбранном для представления ключевом поле или полях не гарантируется уникальность значения, то данные могут быть изменены вместо одной сразу в нескольких записях.
- Значение **Key And Updatable Fields** приводит к поиску записи по тем полям, которые вы пометите как модифицируемые или разрешенные для изменений и по ключевым полям.
- Значение **Key And Modified Fields** обеспечивает поиск по ключевым и фактически измененным полям.
- Значение **Key and Timestamp** для локальных представлений не поддерживается.

Последние две кнопки выбора позволяют вам решить, как будут производиться изменения в исходной таблице: путем удаления и вставки новой записи или путем модификации записи. Второй путь предпочтительнее, так как результат достигается много быстрее.

Свойства **WhereType** и **UpdateType** можно установить глобально для всего Visual FoxPro, используя диалоговое окно **Options** на странице **Remote Data**. Там есть два раскрывающихся списка, объединенных общим заголовком **SQL Updates**, - **Criteria** и **Method**, как это показано на рис. 7.5.

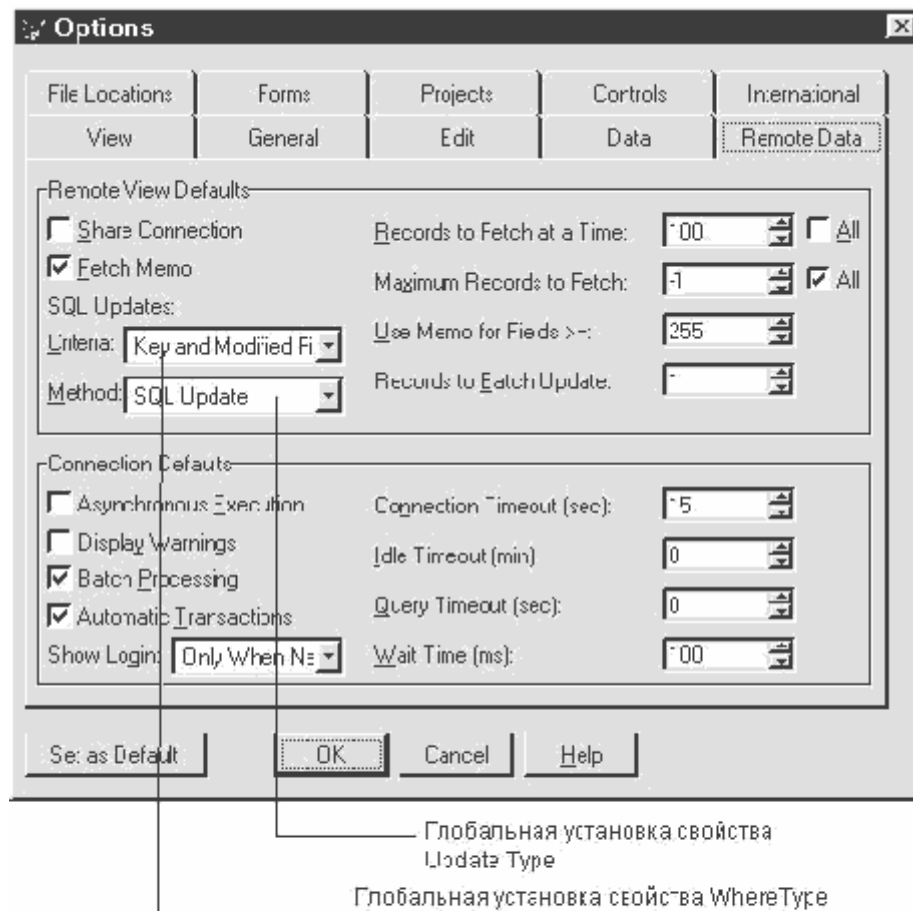


Рис. 7.5. Установка параметров обновления данных

Мы уже говорили, что создать представление можно программным путем с помощью команды **CREATE SQL VIEW**. Свойства, с помощью которых регулируются данные в исходных таблицах, устанавливаются также программным путем с помощью функций **DBSETPROP()** и **CURSORSETPROP()**. Узнать значения установленных свойств можно с помощью функций с похожими названиями: **DBGETPROP()** и **CURSORGETPROP()**. Отличие их в том, что функции **CURSORGETPROP()** и **CURSORSETPROP()** работают с представлениями, которые в данный момент используются в какой-либо из рабочих областей. Функции **DBGETPROP()** и **DBSETPROP()** устанавливают свойства для любого представления, которое содержится в текущей базе данных. В приводимой ниже табл. 7.1 обратите внимание на то, что многие свойства можно установить только программным путем. Свойства, которые приводятся в табл. 7.1, относятся к полям представления. То есть вторым аргументом функции **DBSETPROP()** или функции **DBGETPROP()** будет **"FIELD"**. Например:

```
DBSETPROP("account_and_customer.account","FIELD","comment",; "Используйте это поле для ввода номера счета")
```

Обратите особое внимание, что название поля приводится вместе с названием представления "account_and_customer.account". В противном случае Visual FoxPro не сможет найти это поле.

Таблица 7.1 Свойства полей представлений

Свойство	Тип	Описание
Caption	C	Заголовок поля. Доступно для чтения и записи.
Comment	C	Комментарий поля. Доступно для чтения и записи.
DataType	C	Данное свойство игнорируется для локальных представлений. Доступно для чтения-записи при работе с внешними представлениями.
Default value	C	Значение по умолчанию для поля. Доступно для чтения и записи.
KeyField	L	Содержит .T., если поле указано как ключевое индексное выражение. Данное свойство можно установить визуально в Конструкторе представлений на странице Update Criteria. Доступно для чтения и записи.
RuleExpression	C	Правило проверки ввода уровня поля. Доступно для чтения и записи.
RuleText	C	Сообщение, выводимое на экран в случае нарушения правил проверки ввода уровня поля. Доступно для чтения и записи.
Updatable	L	Содержит истину (.T.), если поле доступно для изменений. Это свойство можно установить визуально с помощью Конструктора представлений на странице Update Criteria. Доступно для чтения и записи.
UpdateName	C	Название поля, которое используется, когда поле модифицируется на таблице внешнего формата. По умолчанию совпадает с именем поля во внешней таблице. Доступно для чтения и записи.

В следующей таблице приводятся некоторые свойства представления, которые можно использовать как для локальных, так и внешних представлений. Для их чтения и изменения необходимо использовать функции **DBGETPROP()** и **DBSETPROP()** со вторым аргументом **VIEW**. В следующем примере мы устанавливаем свойство **Comment** для всего представления, а затем выводим его на экран:

```
=DBSETPROP("account_and_customer","view", "Comment",
"Предназначено для корректировки поставок по счетам")
? DBGETPROP("account_and_customer","view", "Comment")
```

Таблица 7.2. Свойства полей для представлений

Свойство	Тип	Описание
Comment	C	Текст комментария представления. Доступно для чтения и записи.
FetchMemo	L	Равняется .T., если данные из полей примечаний или типа General выбираются с результатами представления. Имеет смысл ставить значение

		этого поля в .F. В таком случае данные поля этого типа будут выводиться только при прямом обращении к ним. По умолчанию имеет значение .T.. Доступно для чтения и записи.
MaxRecords	N	Устанавливает максимальное значение, которое выбирается в представлении. По умолчанию равно 1, то есть выбираются все записи. Если установить это значение равным 0, то не будут выводиться никакие результаты. Доступно для чтения и записи.
RuleExpression	C	Правило проверки вводимых данных уровня записи. Доступно для чтения и записи.
RuleText	C	Сообщение, выводимое на экран при нарушении правил проверки ввода уровня записи. Доступно для чтения и записи.
SendUpdates	L	Если установлено в .T., то изменения будут посылаться в исходные таблицы. Доступно для чтения и записи.
SourceType	N	Равно 1, если представление использует только локальные таблицы. Равно 2, если представление использует внешние таблицы. Доступно только для чтения.
SQL	C	Возвращает строку запроса, который выбирает данные для представления. Доступно только для чтения.
Tables	C	Разделенный пробелом список таблиц, участвующих в выборке представления. Доступно только для чтения.
UpdateType	N	Равняется 1, если старые данные в исходных таблицах модифицируются, 2 - если данные вначале удаляются, а затем добавляются. Доступно для чтения и записи.
WhereType	N	Определяет, по какому принципу будет происходить поиск записи в исходных таблицах при модификации данных в представлении. 1 - Только по ключевым полям. Можно использовать DB_KEY из Foxpro.h 2 - По ключевым полям и полям, разрешенным для изменений. Можно использовать DB_KEYANDUPDATABLE из Foxpro.h 3 - По ключевым и измененным полям. Используйте DB_KEYANDMODIFIED из Foxpro.h 4 - Используется только для внешних представлений. Сравнение проводится по полю TIMESTAMP, если оно поддерживается во внешнем представлении.

К открытому в текущий момент представлению для установки его свойств применяется функция **CURSORSETPROP()**. И, соответственно, функция **CURSORGETPROP()** для чтения свойств представления. Обращаем внимание, что эти две функции работают с любыми курсорами, открытыми в любой рабочей области. Курсоры могут отображать как данные из представлений, так и данные из таблиц. При этом таблицы могут быть свободными, то есть не принадлежать никакой из баз данных. Многие свойства можно изменять как с помощью функции **DBSETPROP()**, так и с помощью функции **CURSORSETPROP()**. Есть некоторые свойства, которые характерны только для курсора, такие как Database, которое доступно только для чтения и указывает полный путь к базе данных, служащей контейнером для объекта, данные из которого отображает курсор. Если в свойствах полей представления не указаны ключевые и разрешенные для модификации поля, то можно указать их для уже активного курсора. Например, вы можете проверить для активного курсора представления с помощью функции **CURSORGETPROP()** наличие ключевого поля и в случае отсутствия установить:

```
USE Account_and_customer
If LEN(ALLT(CURSORGETPROP("keyfieldlist")))=0
    =CURSORSETPROP("keyfieldlist",;
```

```
"account.account,customer.name_customer")
ENDIF
```

Аналогичным способом мы можем проверить наличие полей, разрешенных для модификации и переустановить их, помня при этом, что сначала мы должны иметь ключевые поля для каждой таблицы, участвующей в построении представления. Таким образом мы можем дописать предыдущий пример:

```
USE Account_and_customer
IF LEN(ALLT(CURSORMGETPROP("keyfieldlist")))=0
  =CURSORMGETPROP("keyfieldlist",;
"account,name_customer")
ELSE
  IF LEN(ALLT(CURSORMGETPROP("updatablefieldlist"))=0
    =CURSORMGETPROP("updatablefieldlist",;
    "key_customer,selled,summa,;
    name_customer,address")
  ENDIF
ENDIF
```

Но это еще не все, так как необходимо установить свойство UpdatenameList, которое служит для связи между полями в представлении и полями в исходных таблицах. И помимо этого обязательно установите свойство SendUpdates. В итоге для рассматриваемого случая мы получаем следующую процедуру:

```
USE Account_and_customer
IF LEN(ALLT(CURSORMGETPROP("keyfieldlist")))=0
  =CURSORMGETPROP("keyfieldlist",;
"account,name_customer")
ELSE
  IF LEN(ALLT(CURSORMGETPROP("updatablefieldlist"))=0
    =CURSORMGETPROP("updatablefieldlist",;
    "key_customer,selled,summa,;
    name_customer,address")
    =CURSORMGETPROP("updatenamelist",;
    "key_customer account.key_customer,;
selled account.selled,summa account.summa,;
name_customer customer.name_customer,;
address customer.address" )
  ENDIF
ENDIF
=CURSORMGETPROP("SendUpdates",.T.)
```

В следующем примере программным путем создается представление с помощью запроса к двум таблицам: Account и Customer. С помощью функции **DBSETPROP()** назначаются свойства установки ключевого поля и свойства, устанавливающие возможность модификации полей в исходных таблицах.

```
IF NOT DBUSED("auto_store")
  OPEN DATABASE "auto store"
ENDIF
CREATE SQL VIEW Account_and_customer ;
AS SELECT Account.*, Customer.name_customer, ; Customer.address;
FROM "auto store!account", "auto store!customer";
  WHERE Customer.key_customer = Account.key_customer;
  AND Account.selled = .f.
=DBSETPROP("account_and_customer.account",;
  "FIELD","KeyField",.T.)
=DBSETPROP("account_and_customer.name_customer",;
  "FIELD","KeyField",.T.)
=DBSETPROP("account_and_customer.date_write",;
  "FIELD","Updatable",.T.)
=DBSETPROP("account_and_customer.selled",;
  "FIELD","Updatable",.T.)
=DBSETPROP("account_and_customer.summa",;
  "FIELD","Updatable",.T.)
```

```

=DBSETPROP("account_and_customer.address";
    "FIELD","Updatable",.T.)
USE Account_and_customer
=CUSORSETPROP("SendUpdates",.T.)
=CUSORSETPROP("WhereType",3)
GO 3
REPLACE solded WITH .T.
SKIP -1
SELECT Account
BROWSE

```

В Visual FoxPro существует понятие буферизации, установку которой контролирует свойство **Buffering**. Значение этого свойства можно менять с помощью функции **CUSORSETPROP()** и читать с помощью функции **CUSORGETPROP()**.

Свойство **Buffering** может принимать пять значений:

1. - отсутствие какой-либо буферизации.
2. - пессимистическая буферизация записи. При этом значении блокируется редактируемая запись. Блокировка автоматически снимается, и изменения записываются на диск, как только пользователь переходит на другую запись или закрывает таблицу. Другим способом записи изменений на диск может служить использование функции **TABLEUPDATE()**.
3. - оптимистическая буферизация записи. Запись блокируется только в то время, когда она записывается на диск. Для представлений эта блокировка является значением по умолчанию.
4. - пессимистическая буферизация таблицы. Как только вы начинаете редактирование, блокируется вся таблица. При этом запись на диск может произойти только при вызове функции **TABLEUPDATE()** или закрытии таблицы.
5. - оптимистическая буферизация таблицы. Таблица блокируется в момент записи изменений на диск. Для записи изменений на диск надо либо вызвать функцию **TABLEUPDATE()**, либо закрыть таблицу.

7.5. Запросы в Microsoft Access

В этом параграфе вы изучите методику построения разнообразных типов запросов в Access. Узнаете о возможностях визуального инструментария и способах использования для построения запросов макрокоманд и объектов DAO.

Запросы, без преувеличения, являются главным инструментом работы с данными в Access. Бесспорно, вы можете создать одну таблицу с огромным количеством полей, с помощью Мастера изготовить форму для работы с ней и посчитать, что приложение готово. При этом можно сказать, что существует определенный круг задач, для которых такая технология вполне приемлема. Но мы будем рассматривать другие, более распространенные случаи и задачи, которые лучше решать с помощью запросов. И уже на основе запросов в дальнейшем мы будем строить формы.

Access предоставляет нам несколько способов создания запросов. Самый распространенный - создание запросов с помощью Конструктора запросов. Количество видов запросов, которые мы можем создавать с помощью Конструктора, впечатляет и резко выделяет Access среди других продуктов. Из Конструктора запросов легко перейти в режим редактора SQL либо в табличный режим, где мы можем просмотреть результаты запросов. Режим редактора выражений SQL мы будем рассматривать как второй способ создания запросов. Некоторые профессионалы на начальном этапе изучают Конструктор запросов методом "от противного". То есть пишут запрос вручную, а потом выходят в режим Конструктора. Новичкам имеет смысл почаще делать обратную операцию, то есть создавать запросы в Конструкторе и переходить в режим редактора, для того чтобы лучше изучить синтаксис SQL.

Третий способ - это программный способ создания запросов с помощью объектов доступа к данным (DAO). Для этого используется объект **QueryDef**, который и хранит в себе описание SQL запроса.

Последний способ - создание строки запроса и выполнение его с помощью команды **DoCmd.RunSQL**. Этот способ достаточно популярен, при этом база данных становится более компактной, так как описание запроса хранится в виде строки кода, а не в виде описания объекта. Его недостатком является то, что мы не можем получить таким образом объект **Recordset**. В то же время, этот способ прекрасно подходит для создания и выполнения запросов действий (**Update**, **Delete**, **Insert**), так как часто приходится скрывать от пользователя суть динамики изменения наборов данных. Хранение запросов модификации в базе данных, если мы не хотим, чтобы пользователь без необходимости запускал эти запросы, заставляет нас думать об

ограничении доступа. Поэтому логичней хранить запрос в строчках кода.

Для создания запросов с помощью Конструктора необходимо выполнить одну из следующих последовательностей действий:

1. Перейти на страницу Запросы в Контейнере базы данных.
2. Нажать кнопку Создать.
3. В появившемся диалоге выбора таблиц выбрать нужные таблицы.
4. Связать таблицы, если они не связаны, постоянно хранимой связью, зарегистрированной в базе данных.

Либо

1. Выбрать в меню *Вставка* команду *Запросы*. При этом надо помнить, что меню *Access* изменяется в зависимости от того, какой объект активен в данный момент. Если, например, вы находитесь в режиме редактирования таблицы, то меню *Вставка* не содержит пункт *Запросы*.
2. Выполнить шаги 3 и 4 из предыдущей последовательности действий.

При использовании второго способа можно еще более облегчить свой труд. Для этого, находясь на странице Таблицы Контейнера базы данных, выберите таблицу, которую собираетесь использовать в запросе, как это показано на рис. 7.6. После этого выберите команду *Запросы* меню *Вставка*. Вы попадете в Конструктор запросов с уже выбранной таблицей, графический образ которой будет присутствовать в верхней половине Конструктора.

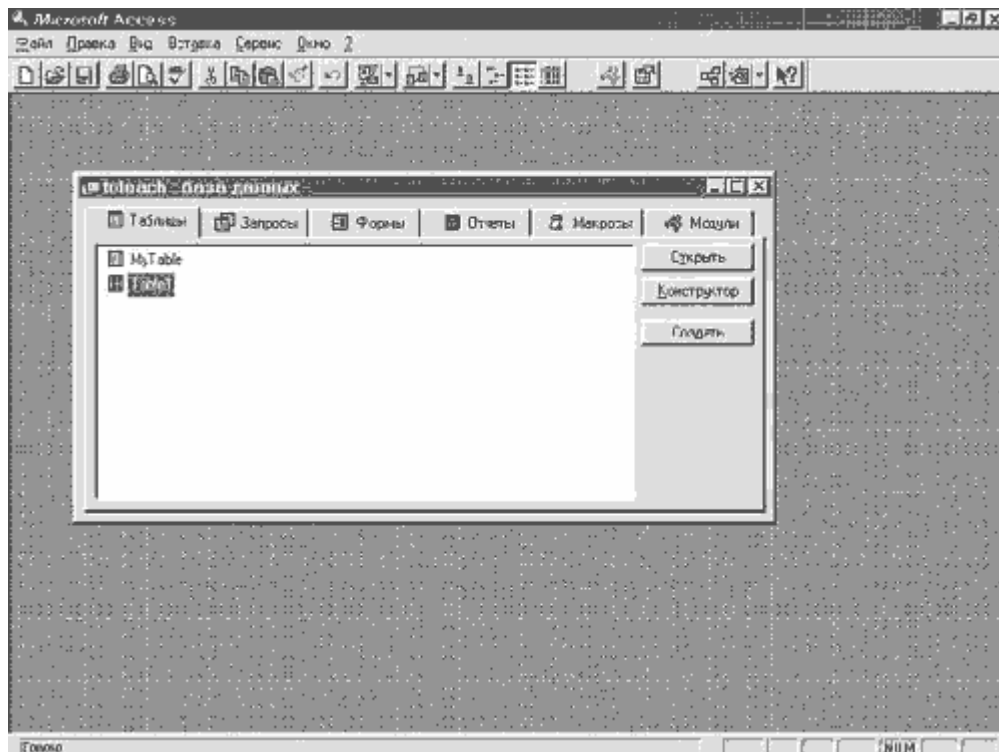


Рис. 7.6.

Здесь необходимо отметить, что запросы можно создавать не только к таблицам, но и к уже существующим запросам. Если есть необходимость, то вы можете комбинировать табличные данные и данные уже существующего запроса в новом запросе. В диалоговом окне *Добавление таблицы* присутствуют три вкладки: *Таблицы*, *Запросы* и *Таблицы и Запросы*. С их помощью вы можете ограничить необходимые вам объекты только таблицами, только запросами, либо, напротив, включить в список как таблицы, так и запросы (рис. 7.7).

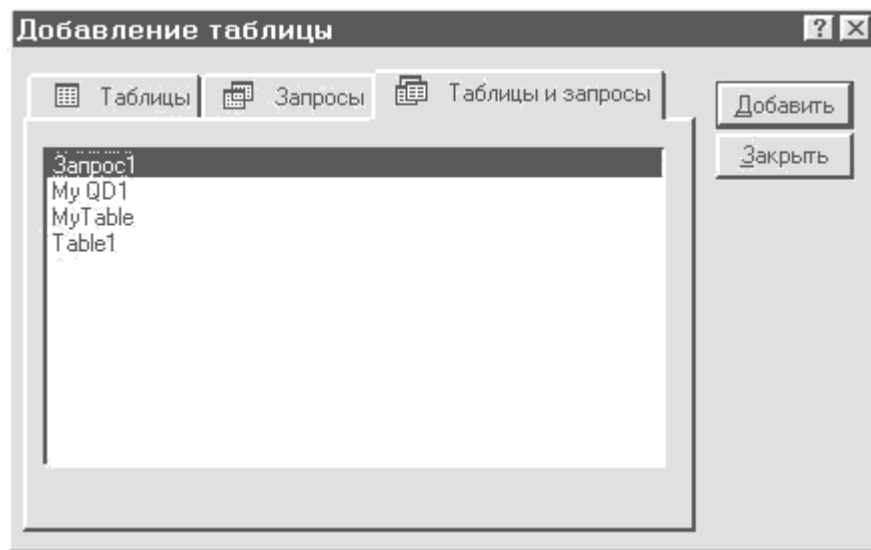


Рис. 7.7.

Так же как в Visual FoxPro, Конструктор запросов состоит из двух частей. В верхней части расположены графические образы таблиц и запросов, данные из которых используются в текущем запросе. По умолчанию Конструктор запросов служит для построения запросов выборки. При этом обязательно учитывайте, что при изменении значений в записях итоговых запросов изменяются данные в исходных таблицах. Нижняя половина представляет собой таблицу, в каждую строку которой мы можем заносить необходимую информацию. Несмотря на то, что научиться пользоваться Конструктором достаточно легко, мы тем не менее позволим себе остановиться на этом моменте более подробно. В первую строку заносится название поля. Сделать это можно несколькими способами:

1. Перетащить графический образ поля из верхней части Конструктора в соответствующую колонку.
2. Дважды щелкнуть мышкой на графическом образе поля.
3. Написать название поля непосредственно в первой строке колонки.
4. Выбрать название поля из раскрывающегося списка, как показано на рис.7.8.

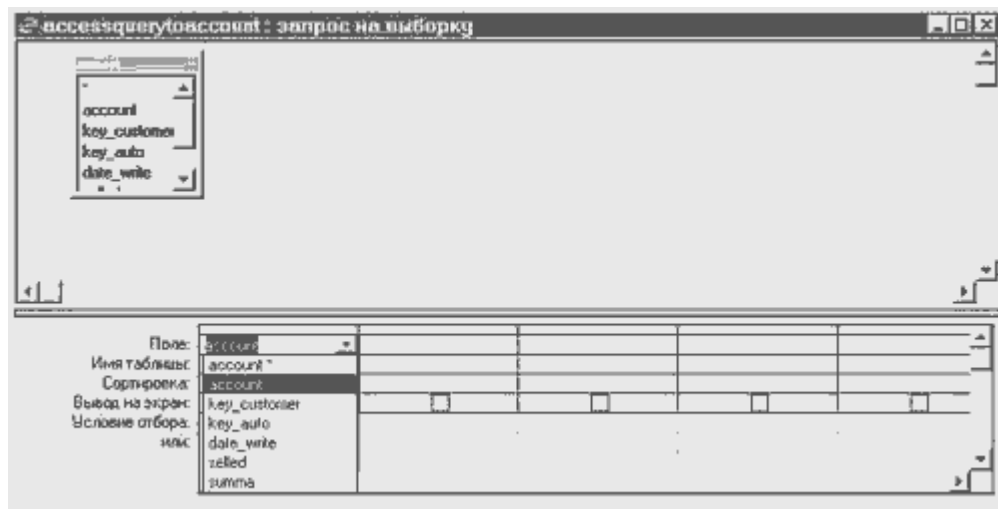


Рис. 7.8.

Во вторую строку заносим название таблицы. Это важно, если у нас больше, чем одна таблица, так как в таблицах поля могут совпадать по названию. Если вы сомневаетесь, что можете правильно занести название таблицы, то выбирайте ее из списка.

Очень часто необходимо выводить информацию, упорядочив ее по какому-нибудь из полей. Для этого используется третья строка колонки поля.

С помощью переключателя, расположенного в четвертой строчке, вы определяете, будет ли поле выводиться в выборке запроса или нет. Представьте ситуацию, в которой вам необходимо отсортировать итоговую выборку по некоему полю, но присутствие этого поля в ней не нужно. Тогда, установив сортировку по этому полю и отключив переключатель вывода на экран, вы

добьетесь необходимой функциональности.

Одним из главных элементов запросов, если не самым главным, является условие выборки или критерии, которые мы накладываем на исходные таблицы. Ведь совершенно нерационально выводить все записи из таблицы, где их десятки тысяч. Поэтому активно используйте критерии для ограничения числа записей. При этом, если в одной колонке вы напишете критерии в строчках "условия отбора" и "или", то они свяжутся по условию **OR**. Если же вам необходимо установить для одного и того же поля критерии по **AND**, то вы должны либо непосредственно набрать условие с использованием этого оператора, либо использовать еще одну колонку для этого поля, но с выключенным переключателем "Вывод на экран", как это показано на рис. 7.9.

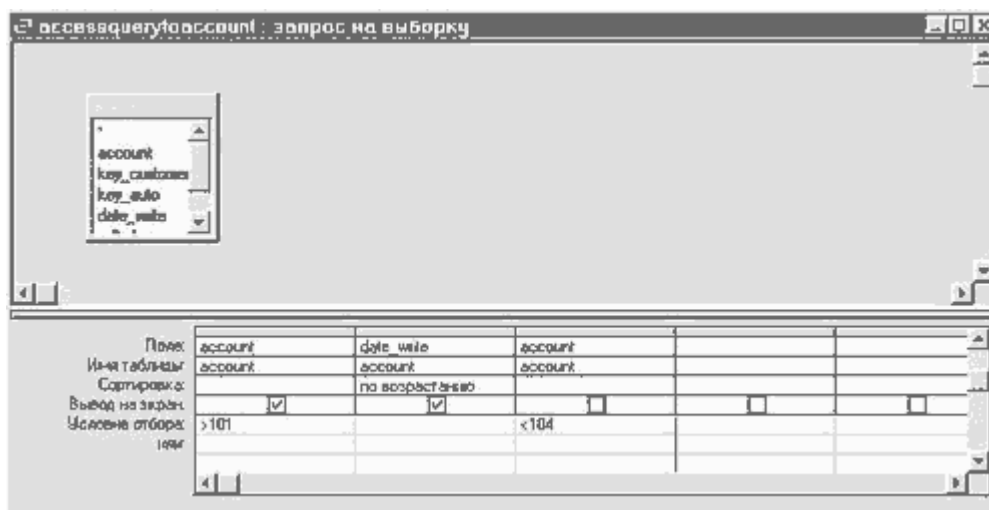


Рис. 7.9. Построение критерия с использованием оператора **AND**. Оба выражения, связанные оператором **AND**, накладываются на одно и то же поле **Account**

Выражения, представленные в табл. 7.3, называются совокупными функциями SQL или функциями агрегирования. Чтобы построить запрос с использованием этих функций, необходимо обязательно установить группировку по одному или более полям. Выберите в меню *Вид* команду *Групповые операции*, либо соответствующий значок на панели инструментов. После того как вы выполните эту команду, в колонках выбора полей появится строчка "Групповая операция". Теперь из раскрывающегося списка вы можете выбрать одну из функций агрегирования для поля, которое отображается в текущей колонке, либо установить группировку по данному полю. Здесь следует обратить внимание на два последних пункта раскрывающегося списка: **Выражение** и **Условие**. Если вы выберете значение **Условие**, то возникнет необходимость отключить вывод на экран этого поля, в противном случае вы получите сообщение от Access, в котором вас будут просить об этом же. Основное назначение этого пункта - занести критерий, который вы укажете в этой колонке, в предложение **WHERE**, так как если вы пишете критерий для поля, по которому происходит группировка, то критерий заносится в предложении **HAVING**.

Таблица 7.3. Функции агрегирования в MS Access	
Имя функции	Операции
Avg()	Среднее арифметическое значений
Count()	Количество записей в наборе
First()	Значение первой записи в наборе
Last()	Значение последней записи в наборе
Min()	Минимальное значение в наборе
Max()	Максимальное значение в наборе
Sum()	Сумма всех записей
StDev()	Стандартное отклонение
StDevP()	Стандартное отклонение смещенное
Var()	Дисперсия
VarP()	Дисперсия смещенная

Иногда бывает недостаточно использовать только функцию агрегирования. Тогда можно использовать выражения, созданные с помощью функций агрегирования для вывода на экран, но при этом в строку *Групповые Операции* необходимо вынести значение "Выражение". На рис. 7.10

приводится пример построения такого выражения. Критерий, указанный для колонки, которая будет выводить выражение, также попадет в предложение **HAVING**.

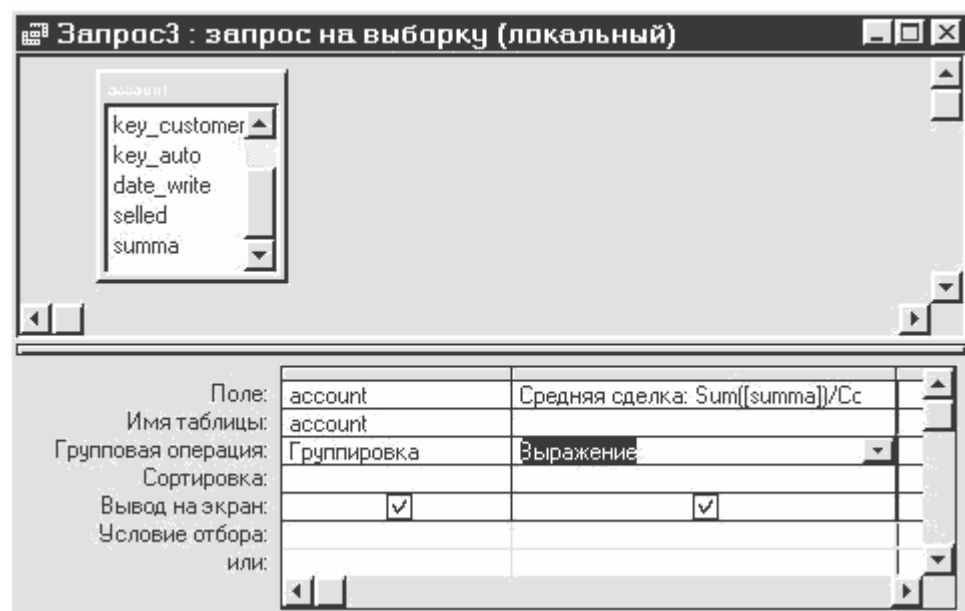


Рис. 7.10.

Чтобы сделать запрос более актуальным, то есть иметь возможность менять содержимое выводимых данных в зависимости от изменяющихся потребностей в получении данных, вы можете использовать параметрические запросы. Параметры запроса устанавливаются с помощью специального диалогового окна, представленного на рис. 7.11. При этом помните, что все выражения тоже будут считаться параметрами, если Access не имеет возможности преобразовать их.

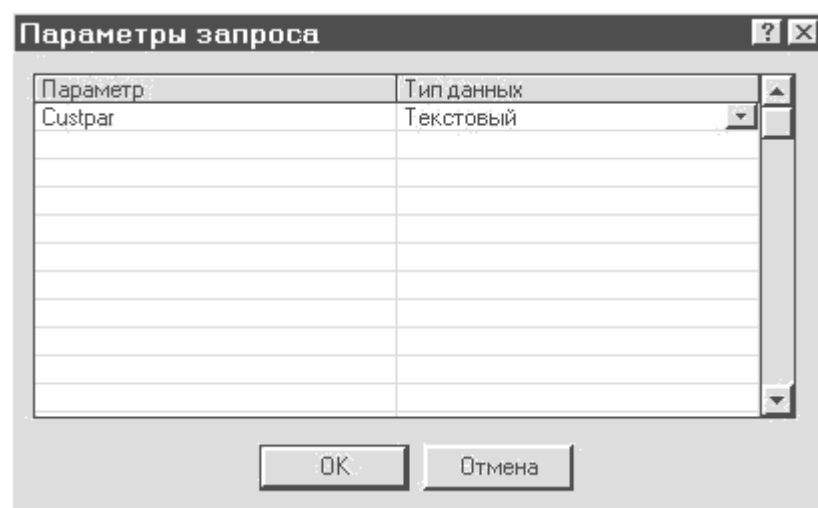


Рис. 7.11.

На рис. 7.12 представлен запрос, который должен вызываться из формы **IKNOWWHATIWANTIOWANTITNOW** и выводить данные по счетам, которые являются текущими для данной формы. В случае запуска этого запроса в момент, когда эта форма не является активной, условие критерия

Forms![IknowwhatIwantIwantitnow].[account]

превратится в параметр, и перед вашими пользователями появится диалог, в котором будет предложено ввести значение данного параметра. В то же время параметр **SKOLKO** зарегистрирован как параметр и всегда будет считаться параметром, даже при наличии поля с одноименным названием в какой-нибудь из форм, из которой будет вызываться запрос.

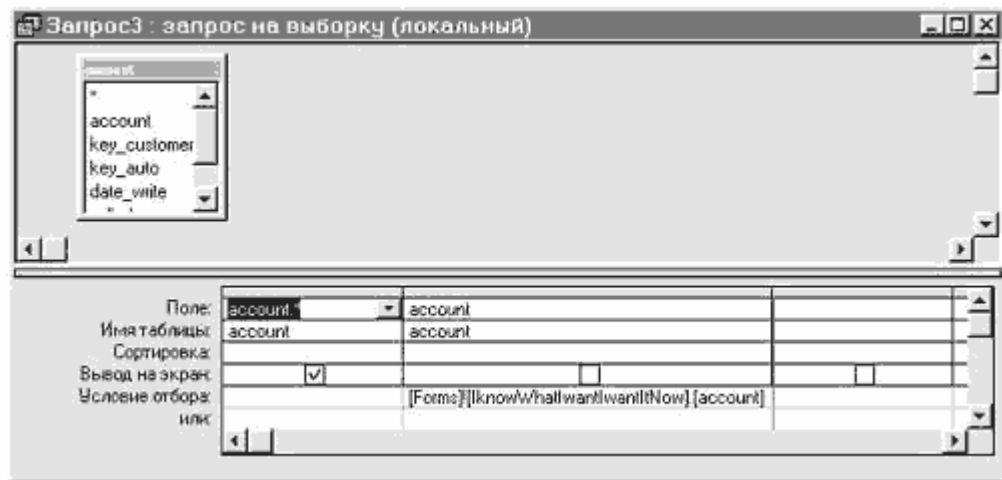


Рис. 7.12. Параметрический запрос, требующий наличия открытой формы

В любой момент мы можем переключиться из окна Конструктора запросов в режим таблицы или режим SQL. Это очень удобно, так как часто мы знаем, что хотим получить, но никак не можем добиться нужного результата из-за неправильной логики запроса. Используя режим таблицы, мы можем корректировать наш запрос до тех пор, пока не получим нужный результат. Режим SQL полезен в плане изучения синтаксиса, кроме того, что очень удобно, если вы правильно отредактируете запрос, то ваши изменения отразятся в окне Конструктора. Существует определенный набор запросов, которые невозможно построить с помощью Конструктора. О них будет упомянуто ниже.

Помимо запросов выборки с помощью Конструктора, можно создать запросы добавления, обновления, удаления, перекрестный и создания таблиц.

Запрос добавления

Запросы добавления - это те запросы, которые начинаются с ключевого слова **INSERT**. Мы их обсуждали выше, теперь рассмотрим, как их создавать визуально в Microsoft Access. По умолчанию Конструктор создает запросы добавления с помощью предложения **SELECT**, то есть использует следующий синтаксис:

```
INSERT INTO назначение [IN внешняя_база_данных]
[(поле1[,поле2[, ...]])]
SELECT [источник.]поле1[,поле2[, ...]]
FROM выражение
```

Если вы в режиме SQL создадите запрос, который использует синтаксис добавления одной записи, то при повторном открытии запроса он все равно преобразуется в запрос с вышеприведенным синтаксисом. Тем не менее на результаты запроса добавления данное преобразование никак не повлияет. Рассмотрим следующий пример. В режиме SQL был создан следующий запрос:

```
INSERT INTO Account ( account ) VALUES (109)
```

После перехода в режим Конструктора и возврат в режим SQL запрос был преобразован к следующему виду:

```
INSERT INTO Account ( account )
SELECT 109 AS Выражение1;
```

Тем не менее и тот другой запрос выполняет совершенно одинаковые действия - добавляет запись в таблицу Account, у которой поле Account имеет значение 109.

Для того чтобы перевести ваш запрос в запрос добавления, выберите в меню *Запросы* команду *Добавление*, либо выберите соответствующий значок на панели инструментов Тип запроса. После этого появится диалог, где будет предложено выбрать таблицу, в которую вы сможете добавить одну или несколько записей, как это показано на рис. 7.13.

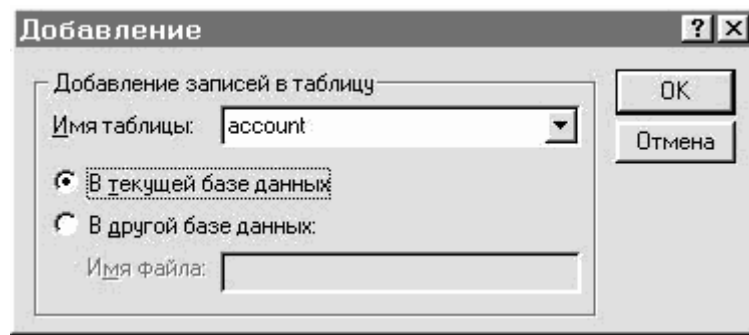
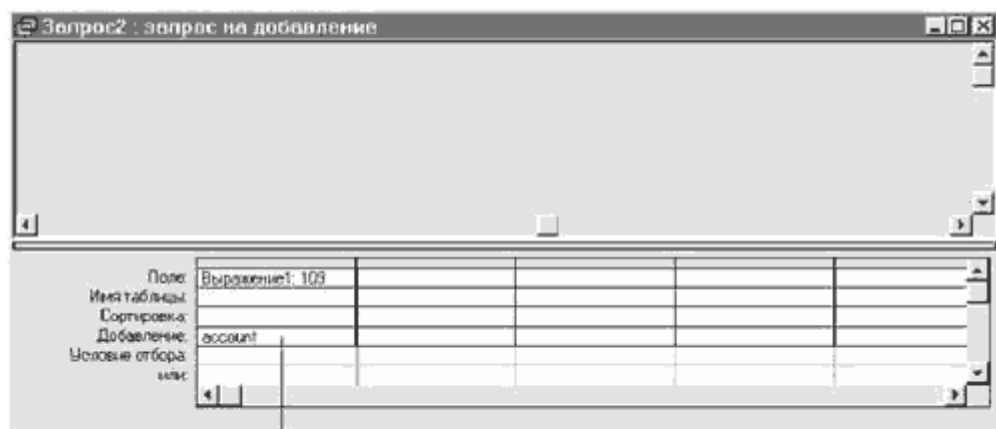


Рис. 7.13. Выбор таблицы для добавления записей

Обратите внимание, что таблица может находиться в другой базе данных, правда, для этого вам еще надо безошибочно указать название файла и полный путь к нему без использования режима Обзор.

После этого Конструктор немного преобразится. Строка Вывод на экране станет называться Добавление, и в ней необходимо указывать поле, в которое вы будете добавлять значения из текущей колонки, как это показано на рис. 7.14.



Строка Добавление заменила строку Вывод на экран

Рис. 7.14.

Запрос - Создание таблицы

Для того чтобы создать запрос данного типа, необходимо выбрать в меню *Запросы* команду *Создание таблицы*. Так же как и в предыдущем типе запросов, вам будет предложено выбрать или ввести название таблицы, причем опять же она может находиться в другой базе данных. По синтаксису полученный запрос очень похож на запрос выборки, только в конце добавляется конструкция следующего вида:

...INTO <<название новой таблицы>>

Так что если вы научились строить запросы выборки, то можете считать, что вы можете строить запросы для создания таблиц.

Запрос удаления

Выбрав команду *Удаление* из меню *Запросы*, вы можете приступить к конструированию запроса удаления. Внешний вид окна Конструктора несколько изменится. Вместо строк *Сортировка* и *Вывод на экран* теперь вы увидите только одну - *Удаление*, которая может принимать два значения, причем без активного участия с вашей стороны. Если выбрать вместо названия конкретного поля звездочку, то строка удаления примет значение "Из". При выборе же конкретного поля строка примет значение "Условие". Причем надо отметить, что если вы напишете условия отбора записей в колонке, где выбрана звездочка, то есть выбраны все поля, то при попытке выполнения запроса появится сообщение о том, что писать эти условия в этой

колонке нельзя. То есть строка Удаления имеет направляющий смысл, никак не влияющий на сам механизм выборки данных (рис. 7.15).



Рис. 7.15.

Запрос обновления

Запрос обновления, для создания которого необходимо повторить те же шаги, что и для предыдущих типов запросов, служит, как видно из его названия, для обновления данных в таблице или нескольких таблицах. Если сравнивать внешний вид Конструктора с предыдущим типом запросов, то вы увидите только одно изменение - строка Удаление заменилась строкой Обновление. В эту строку вам необходимо заносить новое значение поля, которое оно приобретет после выполнения запроса.

Перекрестный запрос

Это последний из типов запросов, которые мы можем создавать визуально. Для этого запроса вы должны выбрать три поля, одно из которых будет содержимым первой колонки полученного набора данных, значения второго поля станут заголовками остальных колонок, а содержимое третьего поля после обработки какой-либо из функций агрегирования будет отображаться во всех остальных колонках.

В качестве примера рассмотрим достаточно тривиальную выборку по трем полям `account`, `key_customer` и `summa`.

Итоговый запрос будет содержать следующие данные:

account	key_customer	summa
101	1	10000
102	2	12300
103	3	13000
104	4	24000
104	4	31000
105	5	34000
105	5	36000
106	6	24000
106	6	28000

Теперь с помощью этих полей построим перекрестный запрос. Для этого вначале, как вы догадались, установим тип запроса с помощью меню или соответствующего значка. Далее обратимся после выбора необходимых нам полей к третьей строке.

Мы можем выбрать для каждой колонки значение в этой строке из набора, который состоит из слов "Группировка", "Выражение", "Условие" и набора уже знакомых нам функций агрегирования. Выбор значений в этой строке накладывает определенные обязанности по выбору значений в четвертой строке, так как Access контролирует ошибки только при попытке

сделать выборку. Иногда бывает очень жаль потраченных усилий.

Таким образом, если мы выбрали для поля account значение "Группировка", то значением строки "Перекрестная таблица" лучше выбрать "Заголовки строк" или "Заголовки столбцов", либо вообще не выводить его, но при этом обязательно добавить еще одно поле, в котором значение строки "Групповая операция" должно быть "Группировка". В нашем примере, как видно из рис. 7.16, мы выбрали значение "Заголовки строк".



Рис. 7.16.

Для поля key_customer выбираем пару значений соответственно "Группировка/Заголовки столбцов". Поле summa будет давать нам информацию о сумме покупок по конкретному счету, поэтому групповой операцией для него станет функция агрегирования **SUM()**, а в перекрестной таблице оно будет представлять собой искомое значение (см. рис. 7.16).

В итоге у нас получится следующий результат.

Account	1	2	3	4	5	6
101	10000					
102		12300				
103			13000			
104				55000		
105					70000	
106						52000

Обратите внимание на полученное выражение. Ключевые слова **TRANSFORM** и **PIVOT** не поддерживаются в стандартном SQL. Поэтому следующий ниже запрос или ему подобные вы можете построить только в Access.

```
TRANSFORM SUM(Account.summa) AS Sum_summa
SELECT Account.account
FROM Account
GROUP BY Account.account
PIVOT Account.key_customer;
```

До сих пор мы рассматривали типы запросов, которые легко создать как визуально, так и в окне режима SQL. Но есть несколько типов запросов, которые невозможно создать с помощью Конструктора. В меню *Запросы* есть пункт *Запрос SQL*, который дает нам доступ к трем командам.

Команда объединения позволяет нам создавать запрос, который будет создавать результирующий набор данных на основе результатов двух запросов. Например:

```
SELECT first,second
FROM tableone
UNION
SELECT first,second
FROM tabletwo
```

В вышеприведенном примере две таблицы имеют поля одинакового типа (непременное условие), поэтому поочередное выполнение двух запросов и объединение их результатов с помощью оператора **UNION** дает нам вполне легкое решение, которое и выполняется достаточно

быстро. В случае отсутствия поля подходящего типа в одной из таблиц запрос можно переписать следующим образом (предполагается, что поле `second` имеет тип `Long`):

```
SELECT first,second
From tableone
UNION
SELECT first,0
FROM tabletwo
```

Как видите, достаточно просто подставить константу подходящего типа.

Следующую команду - *К серверу* (в англоязычной версии *Pass-Through*) мы рассмотрим в [главе 8](#).

Команда *Управление* позволяет нам создавать SQL выражения, которые относятся к DDL разделу SQL.

С помощью Jet SQL нельзя создавать базы данных, но можно создавать таблицы и индексы, а также изменять структуры таблиц и удалять таблицы и индексы.

До сих пор мы использовали визуальные средства и режим SQL для создания запросов. Но очень часто, особенно если вы обращаетесь к базе данных Access из среды приложения, написанного на другом языке, например на Visual Basic или Visual C++, вам придется использовать объекты доступа к данным (DAO).

Типичный запрос с использованием этих объектов выглядит следующим образом:

```
Dim db As Database, qd As QueryDef, mySQL As String
Set db = DBEngine.Workspaces(0).Databases(0)
mySQL = "SELECT * FROM Account WHERE account=102"
Set qd = db.CreateQueryDef("Mywritebyhandquery",mySQL)
```

После выполнения данной процедуры появится запрос, который будет храниться в той базе данных, для которой эта процедура будет текущей. В целом создание запросов с помощью DAO мало отличается от написания запросов с помощью SQL режима. Главной составной частью является второй аргумент метода `CreateQueryDef` - строка SQL.

Для того чтобы обратиться к ранее созданному запросу в программе, можно обратиться к коллекции `QueryDefs`. Вы можете переписать свойство `SQL` объекта типа `QueryDef`, то есть фактически переписать содержание запроса, узнать дату последней модификации запроса и т. д. При этом мы просто обращаемся к свойствам и методам объекта `QueryDef`. В следующем примере для уже существующего объекта переписывается свойство `SQL`, и с помощью окна сообщений выводится дата последней модификации этого объекта, которая, естественно, будет совпадать с текущей системной датой.

```
Dim db As DATABASE, qd As QueryDef, mySQL As String
Set db = DBEngine.Workspaces(0).Databases(0)
Set qd = db.QueryDefs("Simplequery")
qd.SQL = "SELECT * FROM account WHERE date_write >> _
" _ & "#01/06/96#"
MsgBox(Str(qd.LastUpdated))
```

В расположенном ниже примере создается запрос определения данных с помощью метода `CreateQueryDef`, затем он запускается на выполнение. В итоге в текущей базе данных появляется новая таблица `FROMQUERY`.

```
Dim db As DATABASE, qd As QueryDef, mySQL As String
Set db = DBEngine.Workspaces(0).Databases(0)
Set qd = db.CreateQueryDef("DLquery", _
"CREATE TABLE _ fromquery (mama TEXT(20), _
slava LONG)")
qd.Execute
```

Последним способом, который позволяет нам выполнять запросы в Access, являются макрокоманда `RunSQL` или метод `RunSQL` объекта `DoCmd`. В табл. 7.4 представлены все виды запросов, которые мы можем запустить с помощью метода `RunSQL` и ее макроэквивалента `RunSQL`. В дальнейшем мы будем говорить только о методе `RunSQL`.

Таблица 7.4. Запросы, доступные для метода `RunSQL`

Тип запроса	Инструкция SQL
Запрос на изменение	
На добавление	INSERT INTO
На удаление	DELETE
На создание таблицы	SELECT...INTO
На обновление	UPDATE
Управляющий (запрос SQL)	
На создание таблицы	CREATE TABLE
На изменение таблицы	ALTER TABLE
На удаление таблицы	DROP TABLE
На создание индекса	CREATE INDEX
На удаление индекса	DROP INDEX

Синтаксис использования метода RunSQL выглядит так:
DoCmd.RunSQL *инструкция SQL*

Инструкция SQL - это строковое выражение, которое содержит правильное SQL выражение. При этом вы можете обращаться к другой базе данных. Максимальная длина строки - 32768 символов.

В следующем примере требуется наличие формы с названием "Моя форма", которая имеет следующие объекты:

- Несвязанное текстовое поле txtField1.
- Несвязанное текстовое поле txtField2.
- Командная кнопка cmbInsert3.

В методе для события Click запишем следующий код, предполагая, что данные, которые мы занесем в поля txtField1 и txtField2, должны попасть в новую запись таблицы "Моя Таблица":

```
Sub cmbInsert3_Click()
    Dim SQLstr As String, SQLstr1 As String
    SQLstr = "INSERT INTO [Моя Таблица] (Фамилия, Имя) "
    SQLstr1 = "VALUES ([txtField1],[txtField2])"
    DoCmd.RunSQL SQLstr & SQLstr1
End Sub
```

Как вы видите, нам совершенно не нужно иметь отдельный запрос добавления, достаточно написать четыре строчки кода. Теперь пользователь будет добавлять новые записи там, где это выглядит вполне логично, а не экспериментируя с запросами на вкладке Запросы в контейнере базы данных. Данные строчки являются не более, чем примером, никто не пытается призывать вас отказаться от обычного ввода данных посредством использования связанных полей в форме и соответственно от всех средств, которые предоставляет это мощное средство работы с данными.

В этой главе мы пытались показать, насколько удобно использовать SQL - структурный язык запросов, который на текущий момент является самым распространенным средством работы с базами данных. Бесспорно и в Visual FoxPro и в Access вы можете при желании обойтись без SQL, но будет ли это разумно? Чем больше вы используете SQL, тем легче вам переходить с одной СУБД на другую. А многие средства быстрой разработки приложений, такие как Delphi или Visual Basic, просто невозможно использовать без SQL. В этой главе мы не стали рассматривать использование SQL в Visual Basic, но можете считать, что вы уже знаете достаточно, для того чтобы работать с этим средством разработки приложений. Visual Basic использует DAO для работы с базами данных. Следовательно, вы можете использовать объект QueryDef для работы. Можете не откладывать на завтра, а начинать прямо сейчас. Visual Basic, начиная с версии 4.0, имеет еще более гибкое и, в некотором плане, работающее быстрее с данными средство - RDO, но об этом речь впереди.

В [следующей главе](#) мы расширим набор SQL команд, а также рассмотрим вопросы работы с внешними данными.

7.6. Работа с данными в локальной сети

В этом параграфе мы обсудим основные принципы работы в компьютерной сети, которые практически одинаковы во всех системах управления базами данных.

Основное различие, как правило, скрывается в словах рекламных проспектов, которые утверждают, что продукт супернеобычный, сверхскоростной, сам знает, когда запись блокировать, когда разблокировать, или, если угодно, снять блокировку. Некоторые компании поступают еще хитрее - замаскировав новым термином старую технологию ловят доверчивые пользовательские сердца. Ваше дело - верить или не верить рекламе, но разобраться, что необычного нас ждет, когда мы, наконец, решим обрабатывать наши данные в сети, стоит.

При чтении этого параграфа вы

- ознакомитесь с основными особенностями разработки пользовательского приложения, которое предназначено для работы в локальной сети.
- подготовитесь к решению возможных сетевых конфликтов, которые могут возникнуть у пользователей при работе с приложением в сети.

Visual FoxPro

Как правило, данные необходимо использовать не на одном компьютере. Обычно с ними работает группа пользователей. Способ обмена данными посредством дискеты, безусловно, значительно облегчает работу программиста, однако такой способ организации выглядит в настоящее время скорее вызовом, чем правилом. Тем более, что сама работа в сети экономит массу средств и самое главное - время, которое, как вам уже известно, все больше становится эквивалентом денег.

Visual FoxPro обеспечивает работу с данными с помощью монопольного или раздельного доступа к данным, опций блокировки, буферизации таблиц и записей, а также поддержки транзакций. Помимо этого **Visual FoxPro** автоматизирует многоразовую установку среды окружения посредством сессий данных.

В многопользовательских системах необходима уверенность, что только один пользователь может заносить данные в файл или запись в текущий момент. Один из способов добиться этого - открыть таблицу с монопольным доступом. Ни один пользователь или приложение не смогут после этого ни читать данные из этого файла, ни писать в него. Приложение выведет сообщение об ошибке, если какая-нибудь таблица, необходимая для монопольного доступа, уже открыта.

После выполнения команды **SET EXCLUSIVE ON** все таблицы будут открываться в монопольном режиме. Если необходимо проверить текущую установку, то используйте функцию **SET()** с аргументом **EXCLUSIVE** следующим образом:

? SET("EXCLUSIVE")

Если вы не хотите менять текущую установку в приложении для монопольного или многопользовательского режима открытия таблиц, то открывайте таблицу с помощью команды **USE** и точного указания, в каком режиме вы будете ее использовать. Например, если вам обязательно надо открыть таблицу в монопольном режиме, необходимо использовать следующую команду:

USE myfile EXCLUSIVE

В дополнение к монопольному или многопользовательскому режиму доступа к файлам вы можете управлять доступом к таблицам с помощью блокировок как файлов, так и записей. Блокировки могут обеспечить как долгосрочный, так и краткосрочный контроль над данными. Вдобавок, блокировки предотвратят одновременную модификацию двумя или более пользователями одной и той же записи или таблицы. Существуют два типа блокировок: автоматические и ручные.

Если выполняемая команда требует блокировки таблицы или записи и запись или таблица еще не заблокированы, **Visual FoxPro** автоматически пытается совершить блокировку, выполняет команду и снимает блокировку. Команды могут блокировать запись, целую таблицу или заголовок таблицы. Когда заблокирована запись, то другие пользователи могут добавлять, удалять и изменять другие записи, кроме заблокированной. Блокировка таблицы предотвращает любые изменения другими пользователями в таблице. Блокировка заголовка занимает промежуточное положение, другие пользователи могут изменять записи, но не могут удалять и добавлять записи.

В табл. 7.5 приводятся команды, которые осуществляют автоматическую блокировку, и указано, какой вид блокировки они выполняют.

Таблица 7.5. Команды, осуществляющие автоматическую блокировку в Visual FoxPro

Команда	Что блокируется
ALTER TABLE	Вся таблица
APPEND	Вся таблица
APPEND BLANK	Заголовок таблицы
APPEND FROM	Вся таблица
APPEND FROM ARRAY	Заголовок таблицы
APPEND MEMO	Текущая запись
BLANK	Текущая запись
BROWSE, CHANGE, EDIT	Текущая запись и связанные по текущему полю записи в дочерних или родительских таблицах, как только началось редактирование
CURSORSETPROP()	Зависит от параметров
DELETE	Текущая запись
DELETE NEXT 1	Текущая запись
DELETE RECORD < >	Запись < >
DELETE <<область >>	Вся таблица
DELETE-SQL	Текущая запись
GATHER	Текущая запись
INSERT	Вся таблица
INSERT-SQL	Заголовок таблицы
MODIFY MEMO	Текущая запись после начала редактирования
READ	Текущая запись и все записи из других таблиц, связанные по редактируемым полям
RECALL	Текущая запись
RECALL NEXT 1	Текущая запись
RECALL RECORD < >	Запись < >
RECALL <<область>>	Вся таблица
REPLACE	Текущая запись и все записи из других таблиц, связанные по редактируемым полям
REPLACE NEXT 1	Текущая запись и все записи из других таблиц, связанные по редактируемым полям
REPLACE RECORD < >	Запись < > и все записи из других таблиц, связанные по редактируемым полям
REPLACE <<область>>	Вся таблица и все связанные таблицы
SHOW GETS	Текущая запись и все записи из других таблиц, связанные по редактируемым полям
TABLEUPDATE()	Вся таблица
UPDATE	Вся таблица
UPDATE-SQL	Вся таблица

Иногда бывает необходимо применить ручную блокировку. Visual FoxPro блокирует текущую таблицу с помощью функции **FLOCK()**. В отличие от команды **SET EXCLUSIVE ON**, которая предотвращает любой доступ к таблице со стороны других пользователей, **FLOCK()** блокирует таблицу, оставляя ее доступной для чтения. Остальные пользователи могут открывать таблицу и просматривать записи, даже не задумываясь о том, что она заблокирована. **FLOCK()** блокирует текущую таблицу и возвращает .T., если блокировка прошла успешно.

Если в качестве аргумента функции вы укажете псевдоним или номер рабочей области, то у вас появится возможность заблокировать таблицу в другой, не текущей рабочей области. Например: **FLOCK('1')** или **FLOCK('Account')**. Эта особенность очень полезна для блокировки связанных таблиц. **RLOCK()** и **LOCK()** блокируют текущую запись и возвращают .T., если

блокировка завершилась успешно.

Автоматические блокировки удерживаются внутри транзакции до тех пор, пока не произойдут запись или откат на самом верхнем уровне вложения. Ручные блокировки остаются в действии и после завершения транзакции. Единственный способ освободить запись - это снять блокировку вручную. В следующем примере блокируются первые четыре записи:

```
OPEN DATABASE "auto_store"
SET REPROCESS TO 3 AUTOMATIC
STORE '1,2,3,4' TO cRecList
cOldExcl=SET("EXCLUSIVE")
SET EXCLUSIVE OFF
USE model
?LOCK(cRecList,'model')
UNLOCK IN model
SET EXCLUSIVE &cRecList
```

Буферизация защищает данные во время изменений. Буфер автоматически тестирует, блокирует и освобождает записи и таблицы. Существуют два типа буферизации: записи и таблицы, которые защищают такие операции, как модификация данных и их поддержка на уровне одной или нескольких записей.

При использовании буферизации **Visual FoxPro** копирует запись в память или на диск. Первоначальная запись все еще остается доступной для других пользователей. Когда указатель записи перемещается или совершается попытка ее модификации программным путем, **Visual FoxPro** пытается заблокировать запись, проверяет, что другой пользователь не сделал никаких изменений, и затем записывает изменения на диск. Хорошо при этом иметь задействованный обработчик ошибок для разрешения конфликтов, которые могут случиться при попытке записать изменения в таблицу. Буферизация записи отличается от буферизации таблицы тем, что при первом типе изменения записываются в таблицу либо при перемещении указателя, либо при использовании функции **TABLEUPDATE()**. При буферизации таблицы изменения записываются только после использования функции **TABLEUPDATE()**. Буферизация устанавливается с помощью функции **CURSORSETPROP()**. После установки буферизация она остается в действии до ее отключения с помощью той же функции **CURSORSETPROP()**, либо до закрытия таблицы.

Режимы блокировки, используемые с буферизацией, определяют, когда записи блокируются и как блокировка с них снимается. Существуют два режима блокировки: пессимистический и оптимистический.

Пессимистическая блокировка препятствует доступу других пользователей в многопользовательской среде к записи или таблице во время их редактирования. Пессимистический режим блокировки наиболее надежный для изменения индивидуальных записей, но он может замедлить пользовательские операции.

Оптимистическая блокировка - более производительный способ редактирования записей, так как блокировка устанавливается только на момент внесения изменений на диск, что значительно уменьшает время, в течение которого один пользователь монополизирует систему в многопользовательском режиме. Когда вы используете буферизацию для внешних таблиц, **Visual FoxPro** устанавливает оптимистическую блокировку.

Для установки пессимистической блокировки записи используйте функцию **CURSORSETPROP()** и следующие параметры:

```
= CURSORSETPROP("Buffering",2)
```

Visual FoxPro пытается заблокировать запись, на которой установлен указатель. Если блокировка завершилась успешно, **Visual FoxPro** помещает запись в буфер и разрешает редактирование. Когда вы перемещаете указатель записи или выполняете функцию **TABLEUPDATE()**, **Visual FoxPro** записывает данные из буфера в таблицу.

Для установки оптимистической буферизации записи используйте функцию **CURSORSETPROP()** и следующие параметры:

```
= CURSORSETPROP("Buffering",3)
```

Когда вы перемещаете указатель записи или выполняете функцию **TABLEUPDATE()**, **Visual FoxPro** пытается заблокировать запись. Если блокировка успешно завершена, **Visual FoxPro** сравнивает текущее значение записи на диске со значением перед началом буферизации. Если эти значения одинаковые, изменения записываются в таблицу, если же значения разные, то **Visual FoxPro** генерирует ошибку.

Если вы хотите установить пессимистическую блокировку нескольких записей, то используйте функцию **CURSORSETPROP()** следующим образом:

```
= CURSORSETPROP("Buffering",4)
```

В этом случае Visual FoxPro пытается блокировать запись, на которой находится указатель. При успешном завершении блокировки Visual FoxPro помещает запись в буфер и разрешает редактирование. Таким образом вы можете заблокировать несколько записей и все их, естественно, поместить в буфер. После того, как вы используете функцию **TABLEUPDATE()**, все данные из буфера будут переписаны на диск.

Вас может смутить то, что буферизация как правило в документации и литературе упоминается как "буферизации таблицы", а блокируется не таблица, а несколько редактируемых записей. Для проверки вышесказанного используйте возможность Visual FoxPro запускать одну и ту же форму несколько раз в разных сессиях данных. При этом каждую сессию данных вы можете рассматривать как отдельного сетевого пользователя.

Первым делом создайте форму, поместите в нее объект Grid. Теперь для формы установите значение свойства BufferMode равным 1 (pessimistic), а значение свойства DataSession равным 2 (private data session), как это показано на рис. 7.17.

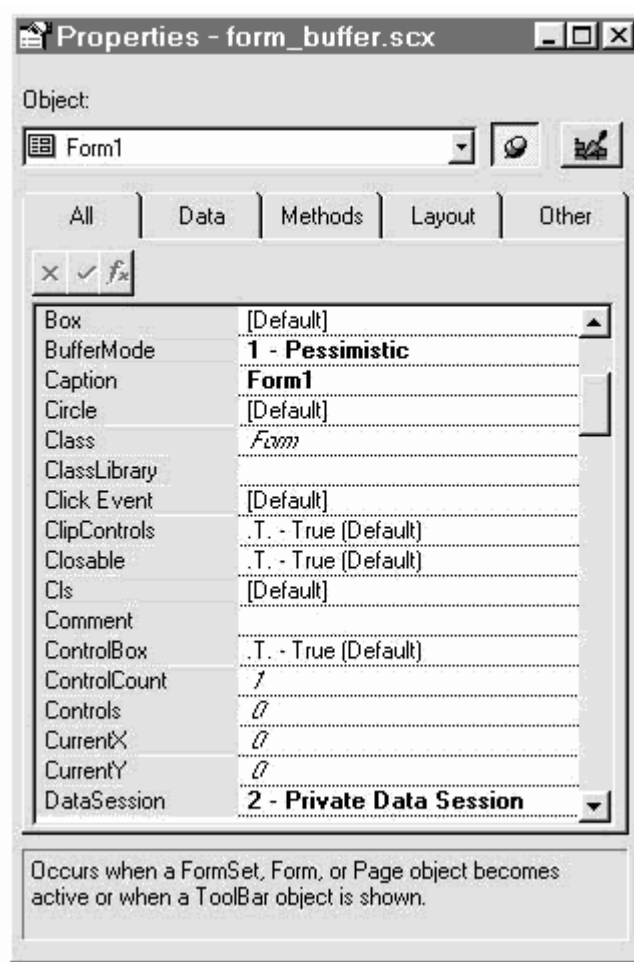


Рис. 7.17.

После сохранения формы запустите ее из командной строки вначале с помощью следующей команды:

```
DO FORM justmadeformforbufferingvalidate NAME f1
```

А затем запустите эту же форму, но уже с другим именем

```
DO FORM justmadeformforbufferingvalidate NAME f2
```

В диалоговом окне **View Window** с помощью списка **CurrentSession** можете проверить, что у вас на данный момент загружено несколько сессий данных, а точнее, одна плюс количество экземпляров нашей формы. Если вы таким же способом загрузите еще одну форму, то количество сессий данных, как вы уже догадались, увеличится на единицу. В окне **View Window**, переключившись в одну из сессий данных, связанных с одним из экземпляров нашей формы, перейдите в рабочую область, в которой открыта таблица, данные из которой отображаются с помощью объекта **Grid**. Нажав на кнопку **Properties**, вы можете увидеть, что буферизация для данной таблицы действительно пессимистическая и множественная (рис. 7.18).

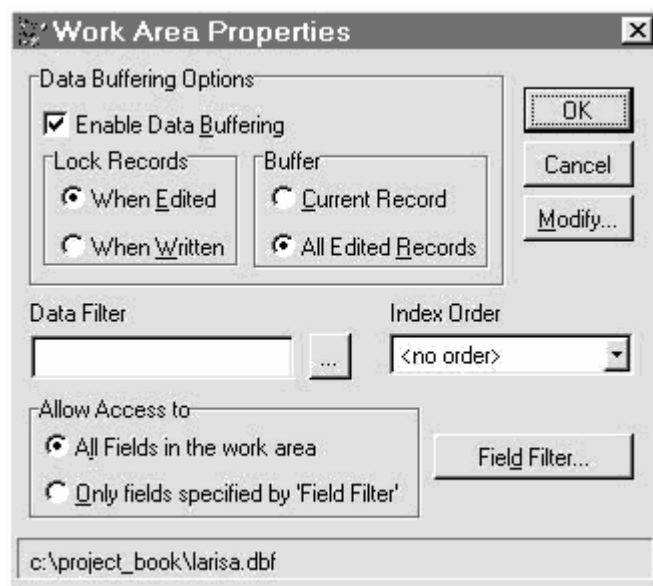


Рис. 7.18.

Теперь в первом экземпляре формы отредактируйте, например, первую и вторую запись, а во втором - третью и четвертую. Как видите, таблицы не заблокированы, но в то же время при попытке отредактировать первую или вторую запись во втором экземпляре вы получите сообщение о том, что запись в данный момент редактируется другим пользователем. И так будет продолжаться до тех пор, пока вы не примените функцию **TABLEUPDATE()** в сессии данных, связанной с первым экземпляром формы, и в рабочей области, где открыта ваша таблица.

Для установки оптимистической блокировки нескольких записей используйте следующее выражение:

```
= CURSORSETPROP("Buffering",5)
```

В этом случае **Visual FoxPro** записывает данные в буфер и разрешает редактирование до тех пор, пока вы не выполните функцию **TABLEUPDATE()**. После этого **Visual FoxPro** совершает следующие действия над каждой записью в буфере:

1. Пытается заблокировать каждую отредактированную запись.
2. После успешной блокировки сравнивает текущее значение на диске со значением до начала буферизации.
3. Записывает данные на диск, если сравнение прошло успешно.
4. Генерирует ошибку, если данные различаются.

Для доступа к данным в буфере можно использовать функцию **RECNO()**. Номер записи из таблицы используется для обращения к ее аналогу в буфере. Если записи добавляются, то при установленной буферизации они вначале попадают в буфер, и номер вновь добавленной записи отрицательный. Отрицательные числа последовательно нарастают по абсолютному значению по мере добавления записей. В табл. 7.6 представлена ситуация, которая может сложиться при добавлении двух записей.

Таблица 7.6. Порядок нумерации записей в буфере

Номер записи	Операция
4	Редактируется
9	Редактируется

12	Редактируется
45	Редактируется
-1	Добавляется
-2	Добавляется

Для того чтобы удалить запись, которая была добавлена, но не записана на диск, вам следует использовать функцию **TABLEREVERT()**, при этом помните, что ее использование с аргументом **.T.** очистит буфер полностью, а с аргументом **.F.** - только от текущей записи. Если вы не выполните функцию **TABLEREVERT()** раньше, чем функцию **TABLEUPDATE()**, то все записи, даже помеченные на удаление, запишутся на диск и при этом будут помечены на удаление.

Мы уже не раз упоминали и использовали в примерах функции **TABLEUPDATE()** и **TABLEREVERT()**. Эти функции можно использовать при установлении буферизации. В противном случае появится системное сообщение о том, что без буферизации их использовать нельзя. Посмотрим более подробно на синтаксис этих функций:

**TABLEUPDATE([IAllRows] [,IForce]] [, cTableAlias
| nWorkArea])**

Первый аргумент **IAllRows** принимает значения **.T.** или **.F.**. Если значение аргумента равно **.T.** и установлена табличная буферизация, то на диск запишутся изменения во всех отредактированных записях, в противном случае запишутся только изменения из текущей записи.

Второй аргумент **IForce**, также принимающий логическое значение, определяет, как относиться к изменениям других пользователей. Если вы вызываете функцию **TABLEUPDATE()** с аргументом **IForce**, равным **.F.**, Visual FoxPro сгенерирует ошибку, как только найдет запись, в которой были сделаны изменения другим пользователем. Вам остается только решить, каким образом поступить с этой записью. В случае, если вы используете функцию **TABLEUPDATE()** со вторым аргументом, равным **.T.**, то все изменения, сделанные другим пользователем или пользователями, будут переписаны.

Если вы работаете с таблицей в текущей рабочей области, то третий аргумент **cTableAlias** или **nWorkArea** можете не указывать. В противном случае надо указать либо псевдоним, либо номер рабочей области таблицы, которую вы редактируете.

Функция **TABLEREVERT()** выполняет обратное по отношению к **TABLEUPDATE()** действие - она очищает буфер без записи изменений на диск.

TABLEREVERT([IAllRows] [, cTableAlias | nWorkArea])

При этом если первый аргумент **IAllRows** равен **.T.**, то очищается весь буфер, если же он равен **.F.**, то буфер очищается только от изменений для текущей таблицы.

Следующие четыре функции помогают находить выход из ситуаций, в которые может попасть приложение, когда велика вероятность одновременного редактирования одних и тех же записей разными пользователями.

Это функции **OLDVAL()**, **CURVAL()**, **GETNEXTMODIFIED()**, **GETFLDSTATE()**. Функция **OLDVAL()** имеет следующий синтаксис:

OLDVAL(cExpression [, cTableAlias | nWorkArea])

Функция **OLDVAL()** возвращает первоначальное значение поля таблицы для текущей записи при установленной буферизации. При этом надо упомянуть, что если таблица в базе данных или курсор имеют правила проверки ввода, то устанавливать буферизацию не обязательно.

Если указатель записи переместится на другую запись, когда установлена буферизация записи, или будет выполнена функция **TABLEUPDATE()** для записи изменений на диск, или будут произведены какие-либо другие действия, которые ведут к модификации данных, первоначальное значение больше доступно не будет.

Тип значения, которое вернет функция **OLDVAL()**, соответствует типу значения аргумента **cExpression**. Например:

```
OPEN DATABASE auto_store
USE account
=CURSORSETPROP("Buffering",5)
GO 3
```

```

REPLACE account WITH 203
?OLDVAL("account")
*** Будет возвращено значение 103
=TABLEUPDATE(.t.)
?OLDVAL("account")
*** Будет возвращено значение 203
Функция CURVAL() имеет следующий синтаксис:
CURVAL(cExpression [, cTableAlias | nWorkArea])

```

Функция **CURVAL()** возвращает значение поля прямо с диска для таблицы или внешнего источника данных.

Значения полей, возвращаемые функциями **CURVAL()** и **OLDVAL()**, можно сравнивать, для того чтобы выяснить, изменял ли другой пользователь в сети значения поля, пока проводилось редактирование данного поля. Естественно, что **CURVAL()** и **OLDVAL()** возвращают разные значения только при использовании оптимистической буферизации записи или таблицы.

Функция **CURVAL()** возвращает значение для текущего поля, и тип возвращаемого значения определяется типом выражения *cExpression*.

В качестве примера слегка увеличим количество объектов в форме **JUSTMADEFORMFORBUFFERINGVALIDATE**. Добавьте два объекта **TextBox**, не связанных ни с каким полем, и две командные кнопки. Для события **AfterRowColChange** объекта **Grid1** напишите следующий код:

```

LPARAMETERS nColIndex
ThisForm.Text1.Value =;
OLDVAL(ThisForm.Grid1.Columns(nColIndex).ControlSource)
ThisForm.Text2.Value =;
CURVAL(ThisForm.Grid1.Columns(nColIndex).ControlSource)

```

Для первой командной кнопки (она будет, к примеру, кнопкой сохранения) напишите всего две строки:

```

=TABLEUPDATE(.F.)
ThisForm.Grid1.SetFocus

```

Вторая кнопка будет отменять изменения, то есть уничтожать содержимое буфера для текущей записи в текущем образце формы:

```

=TABLEREVERT(.F.)
ThisForm.Grid1.SetFocus

```

Теперь можете запустить эту форму несколько раз, проследив при этом, чтобы ее свойство **BufferMode** было равно **Optimistic**, и экспериментировать со значениями в первом и втором текстовом полях. Скорее всего, вы еще раз убедились, насколько хорошо иметь возможность создавать многопользовательскую среду на одном компьютере.

На рис. 7.19 представлена ситуация, когда первоначальное значение поля **RRR** для третьей записи **Greece** в одном экземпляре формы было изменено на **Israel**, а в другом на **Cyprus**.

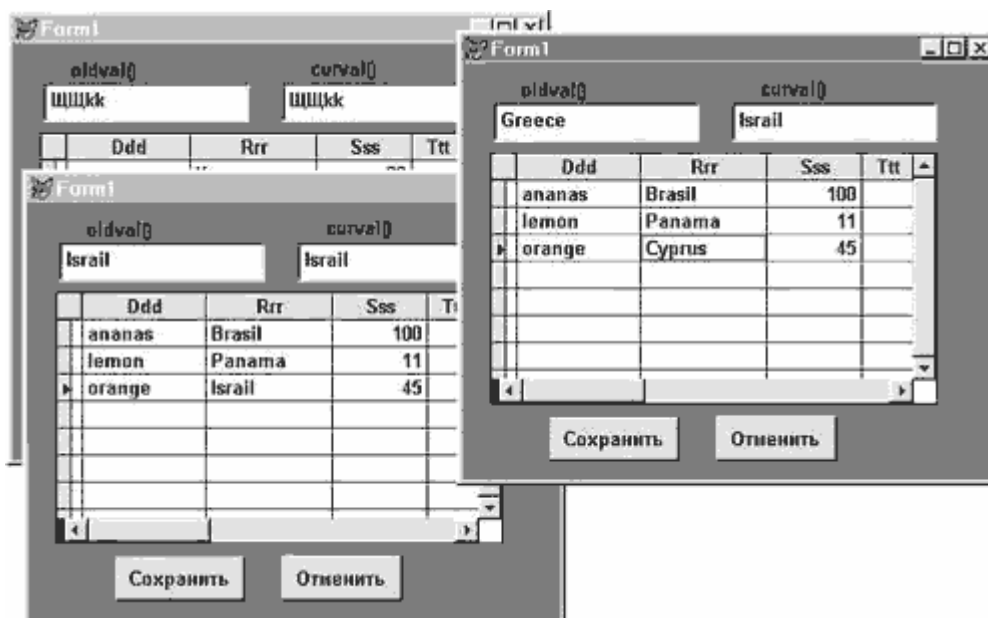


Рис. 7.19.

Так как значение Israel было записано на диск раньше, то теперь в самой правой форме мы имеем разные значения, возвращаемые функциями **OLDVAL()** и **CURVAL()**.

Функция **GETFLDSTATE()** возвращает числовое значение, указывающее, было ли отредактировано значение поля в таблице или курсоре, или была ли запись добавлена, или был ли изменен статус удаления текущей записи. Если вы не указываете псевдоним или номер рабочей области, то функция **GETFLDSTATE()** возвращает значение для поля в текущей таблице или курсоре. Эта функция имеет следующий синтаксис:

GETFLDSTATE(cFieldName | nFieldNumber [, cTableAlias | nWorkArea])

Следует отметить, что **GETFLDSTATE()** только определяет, изменялся ли статус удаления для записи. Например, если вы пометите запись для удаления, а затем выполните команду **RECALL**, то функция **GETFLDSTATE()** укажет, что статус деления изменился, даже если вы вернули его в первоначальное положение. Поэтому мы рекомендуем использовать функцию **DELETED()** для определения статуса удаления.

Функция **GETFLDSTATE()** может употребляться и при отсутствии буферизации. Значения, которые возвращает эта функция, приведены в табл. 7.7.

Таблица 7.7. Возвращаемые значения и соответствующий статус

Значение	Статус
1	Поле не изменялось, и статус удаления не изменялся
2	Поле было отредактировано или статус удаления был изменен
3	Поле в добавленной записи не редактировалось, и его статус удаления не изменилось
4	Поле в добавленной записи изменялось, и его статус удаления изменялся

Функция **GETNEXTMODIFIED()** возвращает номер следующей измененной записи в буферизованной таблице или курсоре:

GETNEXTMODIFIED(nRecordNumber [, cTableAlias | nWorkArea])

Параметр **nRecordNumber** указывает номер записи, начиная с которой необходимо искать запись, подвергнутую изменениям.

Функция **GETNEXTMODIFIED()** возвращает 0, если нет модифицированных записей после записи, которую вы указали. Запись рассматривается как измененная, если содержимое хоть одного из полей было изменено (даже если было возвращено первоначальное значение) или был

изменен статус удаления.

Если какой-нибудь пользователь в сети сделает изменения в буферизованной таблице, любые изменения, которые вы попытаетесь записать на диск с помощью функции **TABLEUPDATE()**, приведут к конфликтной ситуации, кроме, конечно, случая, когда вы используете функцию **TABLEUPDATE()** с двумя аргументами, равными .T. (=TABLEUPDATE(.T.)). Можно разрешить конфликтную ситуацию с помощью функций **OLDVAL()** и **CURVAL()**. Функция **CURVAL()** возвратит значение на диске в текущий момент, а **OLDVAL()** - значение поля записи в момент, когда началась буферизация.

В качестве примера внесем еще несколько изменений в форму **JUSTMADEFORMFORBUFFERVALIDATING**. Добавьте кнопку, к примеру, с заголовком "Все-таки изменить" и сделайте ее изначально недоступной. Код для события Click кнопки "Сохранить" перепишите следующим образом:

```
GO GETNEXTMODIFIED(0)
k=TABLEUPDATE(.F.)
IF k = .F.
    This.Parent.Command3.Enabled=.T.
ENDIF
ThisForm.Grid1.SetFocus
```

Для кода события Click кнопки "Все-таки изменить" напишите следующий код:

```
=TABLEUPDATE(.F.,.T.)
This.Enabled=.F.
ThisForm.Grid1.SetFocus
```

Для события Click кнопки "Отменить" внесите следующие несущественные изменения:

```
=TABLEREVERT(.F.)
ThisForm.Command3.Enabled=.F.
ThisForm.Grid1.SetFocus
```

В итоге у вас получится форма, которая будет работать следующим образом. Когда вы внесете изменения в одни и те же записи и в первом и во втором образце формы, то после нажатия кнопки "Сохранить" вы попадете на первую измененную запись. Если никаких проблем не возникнет, то есть значение **CURVAL()** и **OLDVAL()** совпадет, то значение будет сброшено из буфера на диск. В случае возникновения проблем, если переменная k примет значение .F., то станет доступна кнопка "Все-таки изменить" и при этом в текстовых полях Text1 и Text2 вы увидите и значения, возвращаемые функциями **OLDVAL()** и **CURVAL()**. То есть у вас есть вся информация для принятия решения. При особом желании вы можете модифицировать вашу форму так, что будет возможность при конфликтных ситуациях восстанавливать значение, которое имело поле до начала буферизации, то есть то значение, которое возвратит функция **OLDVAL()**.

Транзакциями в Visual FoxPro называется набор операций, которые:

- изменяют данные, но могут рассматриваться как одна единица;
 - могут управлять конкурирующими изменениями данных;
 - могут использоваться для более легкого управления в целях перехвата ошибок.
-

Транзакции обеспечивают наибольшую защиту существующим данным. Транзакции могут использоваться отдельно, либо вместе с буферизацией записи или таблицы. Только таблицы, которые находятся в базе данных, могут воспользоваться транзакциями.

Visual FoxPro предоставляет три команды, которые обеспечивают контроль за транзакциями:

- **BEGIN TRANSACTION** - предназначена для инициализации транзакции.
- **ROLLBACK** - выполняет откат всех изменений, сделанных после последней команды **BEGIN TRANSACTION**.

- **END TRANSACTION** - блокирует записи, записывает на диск все изменения, сделанные после ближайшей команды **BEGIN TRANSACTION**, затем снимает блокировку с записей.

Изменения в таблицах, в индексных файлах CDX и в полях примечаний таблиц, которые принадлежат базам данных, могут использовать команды транзакций. Транзакции могут работать только с полями таблиц или представлений, но не с переменными памяти и другими объектами.

Изменения выполняются в конце транзакции. Транзакция кэширует изменения на диске или в памяти. Когда транзакция заканчивается, изменения записываются на диск. Если изменения не могут быть записаны на диск, вся транзакция целиком откатывается и ни одно изменение не завершается. Транзакции обеспечивают встроенную систему, которая защищает базу данных от разрушения с помощью отмены всех изменений, в случае если по каким-то причинам изменения не могут быть записаны на диск. Транзакции откатываются путем возвращения всех измененных записей и индексов в первоначальное состояние.

Для большей надежности защиты данных транзакции следует использовать вместе с буферизацией для предотвращения потери данных.

Транзакции могут иметь пять уровней вложенности. Если вы попытаетесь добавить шестой уровень, то будет сгенерирована ошибка.

Когда вы изменяете записи в базе данных, которая является частью транзакции, другие пользователи в сети не могут иметь доступа (чтения и запись) к этим записям, пока не завершится транзакция.

Если другие пользователи в сети пытаются получить доступ к записям, которые вы модифицируете, им придется ждать завершения транзакции. Они будут получать сообщение "Record not available<193>Please Wait" ("Запись не доступна. Подождите, пожалуйста"), пока записи не станут доступны. Поэтому необходимо делать транзакции как можно меньше по длине или проводить транзакции в то время, когда другие пользователи не нуждаются в доступе к данным.

Команда **END TRANSACTION** сохраняет все изменения, сделанные в таблицах, индексных файлах CDX и полях примечаний, и заканчивает транзакцию. Все изменения, сделанные в базе данных между предыдущей командой **BEGIN TRANSACTION** и **END TRANSACTION**, завершаются. Если транзакция является транзакцией первого уровня или единственной транзакцией (то есть транзакцией без вложенности), то изменения записываются на диск.

Если транзакция вложенная, то **END TRANSACTION** переводит все кэшированные изменения на следующий уровень. Вложенные транзакции обладают потенциальной возможностью переписать изменения, сделанные в транзакции на более высоком уровне.

В случае если в это время завершается еще одна транзакция, команда **END TRANSACTION** сгенерирует ошибку. Имеет смысл вставить в цикл команду **END TRANSACTION**, которая будет работать до тех пор, пока транзакция не сможет завершиться.

Для того чтобы сделать откат транзакции, которая началась с помощью команды **BEGIN TRANSACTION**, используйте команду **ROLLBACK**. Эта команда восстановит первоначальное состояние таблиц, индексов и полей примечаний.

Когда вам необходимо выяснить, на каком уровне вложенности вы находитесь в процессе транзакции, используйте функцию **TXNLEVEL()**. Ниже приведен простейший пример ее использования:

```
OPEN DATA auto_store
USE Model
BEGIN TRANSACTION
??TXNLEVEL
**** Будет выведено значение 1, которое равняется
***** текущему уровню транзакции
  BEGIN TRANSACTION
  ?? TXNLEVEL
  **** Будет выведено значение 2, которое равняется
  ***** текущему уровню транзакции
  END TRANSACTION
END TRANSACTION
```

Теперь рассмотрим основные правила работы с транзакциями.

- Транзакции должны объявляться с помощью команды **BEGIN TRANSACTION**. Если будут выполняться команды **END TRANSACTION** или **ROLLBACK** без соответствующей команды **BEGIN TRANSACTION**, то будет сгенерирована ошибка.
- Транзакции действуют, пока не будут выполнены команды **END TRANSACTION** или **ROLLBACK**. Транзакции могут проходить через несколько процедур или функций. Если приложение заканчивается без команды **END TRANSACTION**, то выполняется команда **ROLLBACK**.

- Транзакции используют данные, кэшированные в буфере транзакции, а не данные на диске, для того чтобы использовать самые новые данные.
- Транзакции не могут переписать существующий индексный файл с помощью команды **INDEX**.
- Транзакции могут использоваться только с таблицами, принадлежащими базам данных.

Если вы включили ручную блокировку таблицы или файла во время транзакции с помощью функций **FLOCK()** и **RLOCK()**, то необходимо обязательно снять блокировку. Команда **END TRANSACTION** не снимет блокировку.

Во вложенных транзакциях команды **ROLLBACK** и **END TRANSACTION** работают с изменениями, которые произошли после последней команды **BEGIN TRANSACTION**.

Изменения внутри вложенных транзакций не запишутся на диск, пока не завершится самый верхний уровень, то есть не будет выполнена самая последняя команда **END TRANSACTION**.

Если транзакции выполняются над одними и теми же данными, то преимущество имеет то изменение, которое было выполнено последним, независимо от того, на каком уровне оно находится. Например:

```
USE Account
BEGIN TRANSACTION
BEGIN TRANSACTION
    REPLACE count WITH 103 FOR count=203
END TRANSACTION
REPLACE count WITH 203 FOR count 103
END TRANSACTION
```

В таблице значение поля так и останется равным 203.

Перепишем данный пример так, что последняя команда **REPLACE** будет выполняться перед следующим уровнем вложенности:

```
USE Account
BEGIN TRANSACTION
REPLACE count WITH 203 FOR count 103
BEGIN TRANSACTION
REPLACE count WITH 103 FOR count=203
END TRANSACTION
END TRANSACTION
```

Теперь значение поля останется тем же, что и было - 103.

Несколько советов по увеличению производительности при работе в сети в приложениях Microsoft Visual FoxPro

Если локальная станция имеет достаточно места на жестком диске или достаточно RAM, то вы можете улучшить производительность, разместив временные файлы на локальном диске или на RAM диске. Перенаправление этих файлов на локальный диск или диск RAM увеличивает производительность за счет уменьшения обращения к сетевому диску. Вы можете указать альтернативное местонахождение для этих файлов, включив выражения **EDITWORK**, **SORTWORK**, **PROGWORK** и **TMPFILES** в ваш файл **CONFIG.FPW**.

Если есть возможность отсортировать данные, то имеет смысл это сделать, так как работа с таблицами происходит быстрее, если у вас не включены индексы. То есть используйте поиск с помощью команды, а затем отключайте порядок индекса.

Если есть возможность работать с какими-то файлами монопольно, то используйте эту возможность, так как при монопольном режиме доступ к таблицам осуществляется быстрее.

Чтобы уменьшить вероятность попытки одновременного доступа к записи или таблице, сокращайте время блокировок, что можно осуществить, блокируя записи только при сохранении данных на диск, а не во время их редактирования. Оптимистическая буферизация обеспечивает вам кратчайшее время блокировки.

Microsoft Access

Средства настройки оболочки Access позволяют установить режимы по умолчанию для открытия баз данных и таблиц. Для этого используется команда *Параметры* в меню *Сервис*. После

того как появится диалоговое окно, показанное на рис. 7.20, необходимо перейти на вкладку Другие. После этого, используя набор кнопок "Блокировка по умолчанию", вы можете выбрать необходимый тип блокировки.

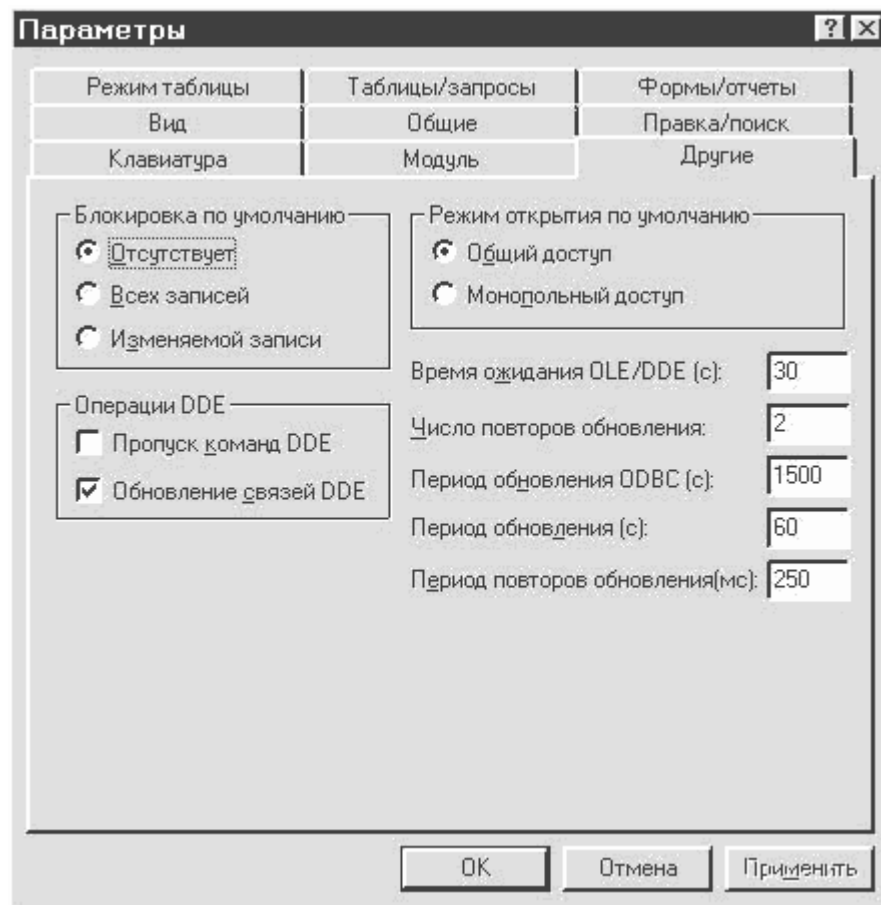


Рис. 7.20.

При выборе кнопки "Отсутствует" вы сможете установить только нежесткую блокировку записи. С помощью такой блокировки все редактируемые записи доступны для других пользователей, но при этом, когда вы отредактируете запись и попытаетесь сохранить изменения, то столкнетесь с одним из двух вариантов. Первый, самый простой: кроме вас запись никто не редактировал и сделанные изменения заносятся на диск. Второй: после того как вы начали редактировать данные, кто-то успел изменить их раньше вас и записать изменения на диск. В последнем случае появится окно сообщения, в котором будет предложено три варианта выхода из сложившейся ситуации: игнорировать чужие изменения и записать на диск свои, скопировать измененную запись в буфер обмена, сохранить чужие изменения и отменить свои. Этот режим иногда называется "оптимистической" блокировкой.

Если вы выберете блокировку изменяемой записи, то для других пользователей будет заблокирована запись, которую вы редактируете, и соседние с ней записи, так как Access устанавливает блокировку для страницы. Страницой для таблиц Access является набор записей размером в 2 килобайта. Эту блокировку иногда называют "пессимистической".

При выборе блокировки всех записей остальные пользователи не смогут открыть таблицу.

При этом следует обратить внимание на тот факт, что если вы работаете с данными с помощью формы, то можете установить для нее блокировки с помощью свойства RecordLocks, которые могут отличаться от установок по умолчанию для текущего сеанса работы Access. Эти типы блокировок строго соответствуют тем видам, которые вы устанавливаете с помощью диалогового окна Параметры.

Свойство RecordLocks помимо форм присутствует еще у запросов и отчетов. При этом для отчетов нет возможности установить блокировку изменяемой записи, скорее всего потому, что отчет их изменять не может.

В диалоговом окне Параметры можно установить режим по умолчанию для открытия баз данных. Но учтите, что режим доступа к базам данных можно выбирать при их открытии с помощью диалога открытия. Если базы данных открываются монопольно, то остальные пользователи не смогут открыть их, поэтому данный режим вряд ли подходит для сетевого использования.

Помимо визуальных средств установки различных режимов блокировки, вы можете использовать объект **Application** для их изменения.

В следующем примере с помощью метода **SetOption** объекта **Application** устанавливается пессимистическая блокировка (блокировка страницы), а с помощью метода **GetOption** и функции **MSGBOX()** выводится номер текущей блокировки.

```
Public Sub mygetoption()
    Dim mystr As String
    Application.SetOption "Блокировка по умолчанию", 2
    mystr = Application.GetOption("Блокировка по_ умолчанию")
    MsgBox (mystr)
End Sub
```

Для форм, отчетов и запросов вы обращаетесь к свойству **RecordLocks**, но учтите, что для уже открытых экземпляров объектов на различных пользовательских машинах свойство не изменится. То есть, изменив свойство некоего объекта, вам попутно надо убедить пользователя закрыть и снова открыть объект, для того чтобы он смог воспользоваться всеми преимуществами нового значения. Пример изменения свойства **Recordlocks** для запроса:

```
Public Sub myqueryproperties()
    Dim db As DATABASE, qd As QueryDef
    Set db = DBEngine.Workspaces(0).Databases(0)
    db.QueryDefs("Моя таблица query").Properties("recordLocks") = 2
End Sub
Пример изменения свойства RecordLocks для формы:
Public Sub ActiveFormRecordLocksChange
    Screen.ActiveForm.RecordLocks = 2
End Sub
```

Учтите, что, как правило, во время работы формы изменять это свойство нет смысла, так как несмотря на то, что вы его установите, форма будет использовать блокировку прежнего типа.

Поэтому рекомендуем следующую последовательность действий для конкретного использования данной процедуры:

1. Создайте функцию, которая будет содержать одну строчку:
ActiveFormRecordLocksChange, то есть вызов процедуры, которая у вас может называться по-другому.
2. Создайте макрос, выполняющий одно действие **RunCode**, аргументом которого будет вызов нашей функции, которая, к примеру, может называться **fActiveFormLocksChange**.
3. Создайте пользовательскую панель инструментов, в которую необходимо перетащить графическое изображение макроса из вкладки Макросы.

Выполнив данную последовательность действий, вы получите пользовательскую панель инструментов с одной кнопкой, которая в режиме Конструктора будет устанавливать для вашей формы свойство **RecordLocks** равным значению 2, что соответствует блокировке изменяемой записи. Проявив фантазию, вы можете дополнить вашу панель инструментов более изощренными инструментами, но основа технологии их создания описана выше.

Очевидно, что то же самое вы можете проделать и для отчетов.

Для того чтобы вы были уверены, что все операции, которые вы хотите провести, были выполнены, используйте методы **BeginTrans**, **CommitTrans** и **RollBack** объекта **Workspace**.

Метод **BeginTrans** начинает транзакцию. Под транзакцией подразумевается серия изменений, которые проводятся над данными и структурой базы данных. Если по какой-либо причине операции, входящие в текущую транзакцию, не могут быть завершены, то система возвращается в исходное состояние. При этом помните, что на рабочей станции должно быть достаточно места на диске, так как при выполнении транзакции вся информация об операциях в нее входящих заносится на диск.

Транзакции должны завершаться с помощью обращения к методу **CommitTrans**. Транзакции могут быть вложенными, не забывайте, что для того, чтобы завершить транзакцию более высокого уровня, вначале необходимо завершить вложенные транзакции. Если по каким-либо причинам приложение не сможет обратиться к методу **CommitTrans**, то система вернется в первоначальное состояние.

Количество вложенных транзакций в **Access**, так же как и в **Visual FoxPro**, не может превышать пяти. При этом обратите внимание на следующий факт. Метод **CommitTrans** для текущего объекта **Workspaces** делает все изменения необратимыми. В то же время, если транзакция вложенная, то

откат транзакции на более высоком уровне приведет систему в первоначальное состояние.

В нижеприведенном примере применяется транзакция для перехода на вторую запись и изменения значения для поля Фамилия. Перед завершением транзакции верхнего уровня предлагается принять решение: заносить изменения на диск или нет.

```
Sub ForceTrans()
    Dim db As DATABASE, wks As Workspace, rst As _ Recordset
    Dim otvet As Integer
    Set wks = DBEngine.Workspaces(0)
    Set db = wks.Databases(0)
    Set rst = db.OpenRecordset("Моя таблица", _ dbOpenDynaset)
    wks.BeginTrans
    wks.BeginTrans
    rst.MoveLast
    rst.AbsolutePosition = 2
    rst.Edit
    rst.ФАМИЛИЯ = "Макашарипов"
    rst.UPDATE
    wks.CommitTrans
    otvet = MsgBox("Изменить", vbYesNo + _ vbDefaultButton1, "Ваше решение")
    If otvet = vbYes Then
        wks.CommitTrans
    Else
        wks.Rollback
    End If
End Sub
```

Кроме вложенных транзакций в Access можно использовать параллельные транзакции. Эти транзакции действуют независимо друг от друга. Но при этом вам необходимо создать еще один объект типа Workspace. Соответственно каждая транзакция завершается независимо друг от друга.

Следующий пример будет работать, только если установлен режим блокировки "Отсутствует". В данном примере обратите внимание на две последние строчки кода. Даже если вы поменяете эти строчки местами, ничего не изменится, то есть вторая транзакция не работает с данными, реально хранящимися на диске.

```
Public Sub multipletrans()
    Dim db As DATABASE, wks As Workspace, rst As _ Recordset
    Dim db1 As DATABASE, wks1 As Workspace, rst1 As _ Recordset
    Set wks = DBEngine.Workspaces(0)
    Set db = wks.Databases(0)
    Set rst = db.OpenRecordset("Моя таблица", _ dbOpenDynaset)
    Set wks1 = DBEngine.Workspaces(0)
    Set db1 = wks1.Databases(0)
    Set rst1 = db1.OpenRecordset("Моя таблица", _ dbOpenDynaset)
    wks.BeginTrans
    rst.FindFirst "[Фамилия]='Клинтон'"
    rst.Edit
    rst.ФАМИЛИЯ = "Доул"
    rst.UPDATE
    wks1.BeginTrans
    rst1.FindFirst "[Фамилия]='Доул'"
    rst1.Edit
    rst1.ФАМИЛИЯ = "Клинтон"
    rst1.UPDATE
    wks.CommitTrans
    wks1.CommitTrans
End Sub
```

Транзакции являются глобальными в рамках объекта Workspace. Поэтому ваши транзакции могут охватывать несколько баз данных и, соответственно, все множество объектов, которые в них содержатся.

Объекты типа Database и Recordset имеют свойство Transactions, которое может принимать значения True или False. От их значений зависит, сможете ли вы использовать методы BeginTrans, CommitTrans и Rollback для работы с этими объектами. Иногда имеет смысл проверять

это свойство перед использованием вышеприведенных методов, особенно при работе с данными из присоединенных таблиц.

Глава 8

Использование технологии клиент-сервер

8.1. Работа с внешними данными с помощью технологии ODBC

Команды Transact-SQL

Создание представлений

Создание триггеров

8.2. Использование Visual FoxPro для разработки клиентского приложения

Синхронный и асинхронный процессы

Создание внешних представлений

8.3. Использование Access и Visual Basic для разработки клиентского приложения

8.4. Использование ODBC API для доступа к внешним данным

8.5. Remote Data Objects

8.6. Внешнее управление сервером с помощью SQL-DMO

До сих пор мы вели речь о приложениях, работающих на одном, локальном компьютере. Если речь шла о компьютерной сети, то БД могла располагаться на файл-сервере или нескольких файл-серверах, в качестве которого может использоваться либо специально выделенный компьютер, либо одна из объединенных в сеть наиболее мощных ПЭВМ. Функции файл-сервера заключаются в основном в хранении БД и обеспечении доступа к ним пользователей, работающих на различных компьютерах. Эти функции обеспечиваются, как правило, той же СУБД, которая работает и на компьютерах пользователей.

При небольших объемах данных эта схема вполне удовлетворяет всем современным требованиям, но с увеличением числа компьютеров в сети или ростом БД начинают возникать проблемы, связанные с резким падением производительности. Это связано с увеличением объема данных, передаваемых по сети, так как вся обработка производится на компьютере пользователя. Если пользователю требуется пара строк из таблицы объемом в сотни тысяч записей, то сначала вся таблица с файл-сервера передается на его компьютер, а затем СУБД отбирает нужные записи. В этом случае длительные перерывы в работе и число выпитых чашек кофе можно сильно сократить, перейдя на технологию клиент-сервер.

Технология клиент-сервер разделяет приложение на две части, используя лучшие качества обеих сторон. **Front-end** (клиентская часть) обеспечивает интерактивный, легкий в использовании, обычно графический интерфейс - находится на компьютере пользователя. **Back-end** (сервер) обеспечивает управление данными, разделение информации, изолированное администрирование и безопасность - находится на специально выделенных компьютерах или даже мейн-фреймах.

При технологии клиент-сервер клиентское приложение (**front-end**) формирует запрос к серверу БД (**back-end**), на котором выполняются все команды. Результаты команд посылаются затем клиенту для использования и просмотра.

Visual FoxPro, Visual Basic и Access обеспечивают средства для создания клиентских частей в приложениях клиент-сервер, которые сочетают мощность, скорость, графический интерфейс, продвинутое средство построения запросов и отчетов.

MS SQL Server является на настоящий момент одним из наиболее мощных серверов БД.

8.1. Работа с внешними данными с помощью технологии ODBC

Важнейшим этапом в построении приложения клиент-сервер является установка связи клиентского приложения с источником данных, находящимся на сервере БД. В настоящий момент различные средства разработки используют несколько технологий обеспечения доступа к данным. Общеизвестным стандартом, как мы уже писали ранее, является технология ODBC.

В этом параграфе мы рассмотрим основы применения технологии ODBC для установки связи с MS SQL Server.

Открытый доступ к данным - **Open Database Connectivity (ODBC)** - это общее определение языка и набор протоколов. ODBC позволяет клиентскому приложению, написанному, например, на Access или Visual FoxPro, работать с командами и функциями, поддерживаемыми сервером.

В качестве сервера может выступать любой сервер БД, имеющий драйвер ODBC (MS SQL Server, Oracle и т. д.), или даже настольная база данных, ведь часто может возникнуть необходимость в совместной обработке данных, хранящихся в формате Paradox, приложениями, написанными и на FoxPro и на Delphi. ODBC находится как бы посередине между приложением, использующим данные, и самими данными, хранящимися в формате, которые мы не можем обработать напрямую в приложении, и используется как средство коммуникации между двумя front-end и back-end сторонами.

Для использования ODBC с целью получения данных в сети вам необходимы средства коммуникации между машиной, на которой находятся данные, и станцией, где они будут просматриваться, модифицироваться и, возможно, пополняться. Это специальная программа, которая называется Администратор ODBC на рабочей станции, и драйверы, которые могут работать как с приложением, так и с данными на сервере. Технология ODBC обеспечивает возможность хранить данные на разных серверах БД, причем не обязательно в одном формате. В теории подразумевается, что для этого не надо переписывать ни одной строчки в вашем пользовательском приложении, работающем на клиент-ской ПЭВМ. На практике так бывает не всегда, из-за различий в структуре различных уровней построения ODBC.

ODBC определяет минимальный набор SQL команд и набор функций вызова с двумя уровнями расширений. Технология ODBC включает также механизм для вызова специфических для сервера возможностей, которые не включены в стандарт ODBC. В целом, это дает возможность разработчику решать самостоятельно, какой уровень функциональности ему достаточен для доступа к серверу баз данных. Разработчик выбирает между наименьшей обеспечиваемой ODBC функциональностью или пытается использовать все возможности сервера.

С минимальным набором функциональности ODBC разработчик может установить связь с источником данных через стандартный интерфейс загрузки, выполнять SQL команды, выбирать данные и получать сообщения об ошибках, если предпринятое действие закончилось неудачей.

С расширенной функциональностью ODBC разработчик может использовать дополнительные возможности стандарта SQL и функции ODBC, чтобы расширить возможности управления внешней СУБД. Расширения ODBC включают в себя такие дополнительные возможности, как асинхронное выполнение запросов, нестандартные в среде приложения типы данных (Timestamp, Binary), прокручиваемые курсоры, SQL команды для скалярных функций, внешние объединения, хранимые процедуры и способность использовать SQL расширения, характерные для конкретного сервера БД.

Если вы используете ODBC драйверы для MS SQL Server, разработанные Microsoft в 1996 году, то вам повезло, так как их расширения предоставляют практически все возможности для полноценной работы с сервером. Если же вы работаете с сервером, отличным от MS SQL Server, или с этим сервером, но драйвер ODBC не разработан фирмой Microsoft, то у вас могут возникнуть трудности ввиду не полной поддержки функциональности сервера драйвером. Если у вас есть сомнение в функциональности драйвера, то для получения полной и содержательной информации о нем используйте функции **SQLGetInfo()**, **SQLGetFunction()** и **SQLGetStmtOption()**. Более подробно о них мы расскажем ниже.

Одна из главных целей создания ODBC - скрыть сложность соединения с сервером и по мере возможности автоматизировать выполнение многочисленных процедур, связанных с получением данных. ODBC требует от разработчика только имени источника данных, при этом функции драйвера, адреса серверов, сети и шлюзы скрыты от пользователя. Возможно, вам как разработчикам захочется знать все тонкости работы ODBC, но учтите, прямая работа с функциями драйвера требует более сложного программирования, что не всегда сопровождается выигрышем в скорости.

Описание основных компонентов ODBC приведено в табл. 8.1.

Таблица 8.1. Основные компоненты ODBC

Компоненты	Описание
Приложение	Использует ODBC для связи с источником данных, отправления и получения данных. Приложение может использовать функцию SQLConnect() , чтобы передать указатель соединения, имя источника данных, идентификатор пользователя и пароль Диспетчеру драйверов.
Диспетчер	Поддерживает связь между

драйверов	приложением и источником данных, обеспечивая информацией приложение и загружая драйверы динамически, по мере необходимости. ODBC позволяет приложению взаимодействовать с источниками данных разных типов посредством драйверов СУБД, используя Администратор ODBC для установки соединения. Диспетчер драйверов и сами драйверы разработаны как библиотеки DLL. Диспетчер драйверов загружает нужную библиотеку, соединяется с сервером с помощью драйвера, который на самом деле и выполняет все вызовы функций из приложения.
Драйвер	Выполняет все вызовы ODBC функций, управляет всеми взаимодействиями между приложением и сервером, переводит SQL выражения на синтаксис источника данных. Драйверы обычно поставляются продавцом сервера БД и имеет смысл получить самую последнюю версию, в которой, вероятнее всего, будет наименьшее количество ошибок. Драйвер перехватывает все ошибки программ, переводя их в стандартные ошибки ODBC. Драйвер позволяет узнать информацию об объектах в базе данных, таких как таблицы, поля, индексы и т. п.
Сервер БД	Хранит и выводит данные в ответ на запросы со стороны ODBC драйвера. Сервер по запросу возвращает данные или код ошибки, которые передаются в приложение.

Первые разработчики ODBC столкнулись с проблемой - какой из диалектов SQL поддерживать. Если есть сотни SQL продуктов на рынке, то есть и сотни диалектов SQL. В итоге, чтобы никого не обидеть, разработчики ODBC создали еще один диалект, который называется "ODBC SQL", и компонент, который позволяет программам переводить их собственные диалекты на универсальный диалект и наоборот.

В первых двух главах мы очень много внимания уделили понятию базы данных как централизованного хранилища всех данных, связанных с функционированием предприятия. Наивно было бы надеяться, что все они всегда будут храниться в одном формате и физически находиться на одной ПЭВМ. Об этом давно догадались сообразительные творцы средств разработки приложений для обработки данных и еще до появления ODBC или параллельно с ним снабдили свои продукты самыми разнообразными технологиями и средствами для доступа к внешним данным.

Поэтому на настоящий момент разработано очень много технологий для доступа к внешним данным. В них немудрено запутаться, и, чтобы этого не произошло, - ни слова о том, что не имеет отношения к рассматриваемым средствам разработки. Кстати, среди них СУБД Access имеет наибольшее число способов взаимодействия с MS SQL Server, поэтому как пример мы хотим кратко описать и перечислить их (см. также рис. 8.1):

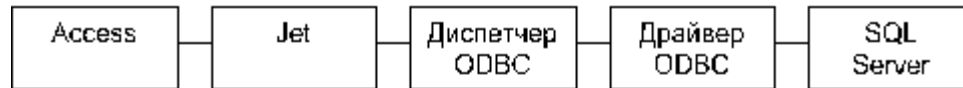
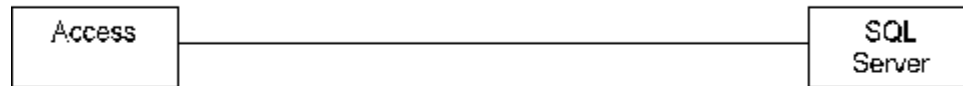
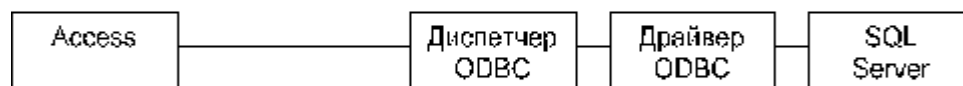
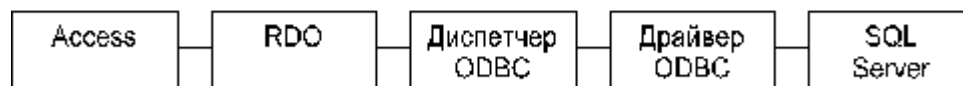
Связанные таблицы*Pass-through запросы**ODBC API**RDO**SQLOLE*

Рис. 8.1.

- **Использование связанных таблиц.** Всю работу за вас выполняет процессор баз данных Microsoft Access. Самый надежный в плане программирования способ - он не требует вообще никаких дополнительных драйверов. Вы можете заниматься только обработкой данных, связывать таблицы на сервере с локальными таблицами, создавать запросы.
- **Использование SQL pass-through функций.** При этом вы обращаетесь напрямую к серверу, на его языке, минуя процессор баз данных Access. Ваш запрос, хотя и проходит через все компоненты ODBC, ни один из них реально не обрабатывает его.
- **Прямое обращение к ODBC API.** При этом способе вы также минуете процессор баз данных Access и напрямую используете Диспетчер драйверов и сам драйвер для доступа к базе данных. Таким образом, ваш код должен обеспечить всю функциональность, которую, как правило обеспечивает процессор баз данных Access. Сложность написания кода даже для довольно простых запросов заставляет обращаться к этому методу только тогда, когда действительно необходима значительная скорость выборки данных.
- **Использование Remote Data Objects.** Этот четвертый способ является самым дорогим по материальным затратам, так как для его функционирования необходимо, чтобы на вашей рабочей станции была установлена версия Enterprise Visual Basic 4.0 и выше. Технология Remote Data Objects (RDO) - Объекты для доступа к внешним данным - трактует запросы ODBC как объекты. Запросы работают со свойствами этих объектов и используют их методы, чтобы получить данные. В свою очередь, RDO конвертирует все действия в вызовы ODBC API. В принципе то же самое делает процессор баз данных, но во многих случаях RDO работает быстрее. Однако RDO также требует написания многих строчек кода, хотя и не такого сложного, как в случае прямого обращения к ODBC API. Ну и вспомните, что установка Enterprise версии Visual Basic стоит около 1000 долларов.
- **Использование SQL DMO.** За счет использования SQL OLE в этом случае не используются ни процессор баз данных, ни ODBC. В данном случае мы работаем с SQL-DMO. В этой технологии мы можем использовать такие продукты, как Visual Basic for Application, Visual Basic (32 bit), Visual FoxPro и C++ (32 bit), для работы с объектами, которые позволяют нам управлять различными SQL серверами в сети. Этот способ работает только под Windows NT и Windows 95. При этом на рабочей станции вам

необходимо иметь клиентскую часть SQL Server. Программирование администрирования SQL Server значительно упрощается благодаря легкому в изучении и использовании объектному языку. Легко можно выполнять все запросы как выборки, так и управления.

После того как вы подсоединитесь к внешнему источнику данных, вы можете задействовать набор SQL - *pass through* функций. С помощью них вы можете выполнять SQL команды, которые обеспечивают следующие операции:

1. Выполнение хранимых процедур.
2. Выполнение запросов на языке сервера.
3. Создание новых баз данных, таблиц и индексов на сервере.
4. Создание и выполнение триггеров, значений по умолчанию, правил проверки ввода и хранимых процедур.
5. Поддержание бюджетов пользователей или выполнение других задач Системного Администратора.
6. Выполнение множественных операций добавления и обновления в одном наборе команд.

Для того чтобы получить доступ ко всем этим возможностям, вам необходимо наличие Администратора ODBC на вашей рабочей станции. Как правило, большинство приложений фирмы Microsoft устанавливают это приложение, если вы при установке указали, что это необходимо. Нужные драйверы, как правило, поставляются с продуктом, доступ к данным которого вы хотите иметь. Помимо этого их выпускают еще третьи фирмы. Можно написать драйвер и самостоятельно, используя ODBC SDK. После того как вы решили все эти проблемы, можете приступать к созданию хранилищ данных на выбранном сервере баз данных.

Команды Transact-SQL

Теперь, когда мы подошли к вопросу создания приложений клиент-сервер, настала пора продолжить изучение следующего набора SQL команд. Помните, что все они доступны из любого средства разработки приложений, которое либо поддерживает ODBC, либо имеет возможность вызывать функции из внешних API.

У нас нет возможности описать все диалекты и нет желания описывать стандартный SQL. Поэтому мы остановимся на MS SQL Server. То есть синтаксис приводимых команд и примеры их использования относятся к MS SQL Server.

Одна из самых сложных команд в Transact-SQL - команда **CREATE TABLE** имеет много опций и позволяет построить настоящую схему данных. Мы рассматривали эту команду в предыдущем разделе, но синтаксис этой команды в Transact-SQL требует более подробного ее рассмотрения в данном диалекте SQL.

Приводим синтаксис этой команды, а в табл. 8.2 описание ее аргументов.

```
CREATE TABLE [database.[owner].]tablename
({colname datatype [NULL | NOT NULL | IDENTITY[(seed, increment)]]
[constraint [ constraint [...constraint]]]
| [[.] constraint]} [[.] {nextcolname | nextconstraint}...])
[ON segmentname]
```

Таблица 8.2. Аргументы команды CREATE TABLE

Аргумент	Назначение
<i>Database</i>	База данных, которая будет содержать таблицу. Если этот параметр опущен, то базой данных по умолчанию будет последняя, открытая с помощью команды USE .
<i>Owner</i>	Владелец новой таблицы, если не указан, то владельцем будет считаться текущий пользователь, который запускает команду.
<i>Tablename</i>	Имя вновь создаваемой таблицы. Должно удовлетворять правилам SQL для идентификаторов.
<i>Colname</i>	Название колонки в таблице. Должно удовлетворять правилам SQL для идентификаторов.
<i>Datatype</i>	Тип данных, поддерживаемый в SQL.

Может быть как встроенным, так и пользовательским типом.

Seed Начальное значение колонки типа **IDENTITY**.

Increment Разница между последовательными значениями в колонке типа **IDENTITY**.

Constraint Ограничения уровня поля или колонки.

Пример использования команды:

```
CREATE TABLE modelSQL
(key_auto smallint IDENTITY(1,1),
model_name varchar(25) NOT NULL,
model_prc money)
```

В этом примере используются следующие опции команды **CREATE TABLE**:

IDENTITY - указывает, что SQL Server автоматически прибавит единицу (аргумент **Increment**), причем отсчет начнется с 1 (**Seed**).

NOT NULL - указывает, что поле должно иметь значения в каждой новой записи. Это устанавливается по умолчанию. Если нужно разрешить полю не иметь значения, то в поле необходимо указать **NULL**.

Каждой колонке должен быть присвоен тип данных, которые она может хранить. В табл. 8.3 мы приведем типы данных, которые поддерживаются в MS SQL Server для колонок.

Таблица 8.3. Допустимые типы данных для MS SQL Server

Тип данных	Описание
binary(n)	Данные бинарного типа длиной ровно n бит. Не может принимать значения типа NULL .
varbinary(n)	Данные двоичного типа длиной до n бит. Может принимать значения типа NULL .
char(n)	Данные символьного типа длиной ровно n символов. Не может принимать значения типа NULL .
varchar(n)	Данные символьного типа длиной до n символов. Может принимать значения типа NULL .
datetime	Тип дата, который может принимать значения между 1 января 1753 и 31 декабря 9999 с точностью в 3.33 миллисекунд.
small-datetime	Тип дата, который может принимать значения между 1 января 1900 и 06 июня 2079 с точностью в 1 минуту.
decimal(p,s)	Десятичное число, которое может иметь всего до p знаков и до s знаков после запятой. Значение p должно быть не больше, чем 38, а s не больше, чем p.
numeric(p,s)	Десятичное число, которое может иметь всего до p знаков и до s знаков после запятой. Значение p должно быть не больше, чем 38, а s не больше, чем p.
float(n)	Число с плавающей запятой с количеством знаков после запятой не больше 15. Позволяет достигнуть точности 10 в 38 степени.
real	Число, которое может иметь до 7 знаков после запятой.
int	Целое число, принимающее значения между -2 147 483 648 и 2 147 483 647.
smallint	Целое число, принимающее значения

	между -32768 и 32767.
tinyint	Целое число, принимающее значения между 0 и 255.
money	Неокругленное число с плавающей запятой от -922 337 203 685 477.5808 до 922 337 203 685 477.5808.
smallmoney	Неокругленное число с плавающей запятой от -214 746.3648 до 214 746.3647.
bit	1 или 0.
Timestamp	Уникальное значение, которое генерирует SQL, каждый раз, когда запись редактируется. Каждая таблица может иметь только одно поле типа Timestamp .
text	Поле, принимающее значения символьного типа длиной до 2 Мб.
image	Поле двоичного типа длиной до 2 Мб.

Помимо приведенного в табл. 8.3 набора типов данных, можно создавать свои типы данных. Для этого используется системная хранимая процедура **sp_addtype**.

В синтаксисе команды **CREATE TABLE** есть слово *constraint* (ограничение). Разрешено пять типов ограничений:

1. Первичный ключ (**Primary Key**) таблицы.
2. Уникальность (**Unique**) таблицы.
3. Ссылка (**Foreign Key**) таблицы.
4. Значение по умолчанию (**Default**) колонки.
5. Правило проверки уровня колонки (**CHECK**).

Чтобы создать первичный ключ, который следит за уникальностью значений по его выражению первичного ключа, включают ограничение первичного ключа. Главное же предназначение первичного ключа - это использование его в декларативной ссылочной целостности, с помощью которой вы можете проводить каскадные изменения данных (модификация, удаление и вставка) в дочерних таблицах, то есть таблицах, связанных с родительской по выражению первичного ключа. Синтаксис этого ограничения выглядит следующим образом:

```
[CONSTRAINT constraintname]
PRIMARY KEY [CLUSTERED | NONCLUSTERED]
(colname [, colname2 [..., colname16]])
[ON segmentname]
```

Первичный ключ может проиндексировать до 16 колонок в таблице. В таблице может быть только один первичный ключ. Первичный ключ может относиться к типу **CLUSTERED** или **NONCLUSTERED**. Тип **CLUSTERED** создает объект, в котором физический порядок записей такой же, как и индексный. Естественно, что это увеличивает скорость поиска записи. Только один индекс типа **CLUSTERED** может присутствовать в таблице. Обратите внимание, что ключевое слово **CONSTRAINT** опциональное. Вы можете именовать ограничение, а можете не именовать. Если вы создадите ограничение с именем, то его имя будет появляться в системных сообщениях в случае, если будет предпринята попытка нарушить наложенное ограничение.

Например:

```
CREATE TABLE Auto_Store.YourReadness.FirstTable
(sqlserverstns CONSTRAINT pk_SQL PRIMARY KEY CLUSTERED,
access_users)
```

или можно так:

```
CREATE TABLE Auto_Store.YourReadness.FirstTable
(sqlserverstns PRIMARY KEY CLUSTERED,
access_users)
```

Как уже указывалось, в таблице может быть только один первичный ключ, но с помощью ключевого слова **UNIQUE** можно подготовить кандидатов на первичный ключ и при необходимости с помощью команды **ALTER TABLE** переустановить первичный ключ. Для создания ключей-кандидатов используется следующий синтаксис:

```
[CONSTRAINT constraintname]
  UNIQUE [CLUSTERED | NONCLUSTERED]
    (colname [, colname2 [..., colname16]])
    [ON segmentname]
```

Например:

```
CREATE TABLE Auto_Store.YourReadness.FirstTable
(sqlserverstns CONSTRAINT pk_SQL PRIMARY KEY CLUSTERED,
access_users CONSTRAINT uk_ac_users UNIQUE)
```

Учтите, что вы можете иметь только один индекс типа **CLUSTERED**, то есть в нашем случае второй индекс обязательно будет **NONCLUSTERED**.

Следующее ограничение служит для построения ссылочной целостности, то есть для сохранения связей между двумя таблицами по выражению, которое мы укажем с помощью ключевого слова **FOREIGN KEY**. При этом те же поля должны быть указаны после ключевого слова **REFERENCES**, где также должна быть указана таблица, с которой организуется связь. Дополнительно необходимо, чтобы в родительской таблице обязательно присутствовал первичный или альтернативный ключ по тем же полям. Синтаксис этого ограничения:

```
[CONSTRAINT constraintname]
  [FOREIGN KEY (colname [, colname2 [..., colname16]])]
  REFERENCES [owner.]reftable [(refcol [, refcol2
    [..., refcol16]])]
```

Исходя из вышесказанного, понятно, что в ключ ссылки (**FOREIGN KEY**) может входить не больше 16 полей. Надо отметить, что ссылки с помощью **FOREIGN KEY** и **REFERENCES** мы можем установить только для таблиц, которые находятся в одной базе данных. В следующем примере обратите внимание на изменение в записи ограничения при построения ключа по нескольким полям. Пример построения ссылочной целостности по одному полю:

```
CREATE TABLE autostore.yourreadness.pat_table
(tree char(20) PRIMARY KEY,
leaf char(10))
CREATE TABLE autostore.yourreadness.child_table
(wood char(20) REFERENCES pat_table(tree))
```

В данном примере нет необходимости указывать ключевое слово **FOREIGN KEY**, так как связь устанавливается по одному полю.

Пример построения ссылочной целостности по нескольким полям:

```
CREATE TABLE autostore.yourreadness.pat_table
(tree char(20),
branch money,
leaf char(10),
CONSTRAINT tochild PRIMARY KEY (tree,branch) )
CREATE TABLE autostore.yourreadness.child_table
(tree char(20),
branch money,
childs varchar(25),
FOREIGN KEY (tree,branch) REFERENCES pat_table(tree,branch))
```

Следует учитывать, что ключевое слово **FOREIGN KEY** не создает индекса. Поэтому для лучшей производительности имеет смысл создавать для дочерней таблицы индекс по выражению, которое будет использоваться в ссылке.

Ограничение **Default** устанавливает значение по умолчанию для колонки. Оно имеет следующий синтаксис:

```
[CONSTRAINT constraintname]
  DEFAULT {constantexpression | niladic-function | NULL}
  [FOR colname]
```

Здесь следует отметить, что можно использовать либо выражение в виде константы, либо функцию (встроенную или пользовательскую), которая не требует аргументов. Совершенно очевидно, что для колонки типа **Timestamp** значение по умолчанию устанавливать нельзя, так же как и для поля **IDENTITY** (счетчик). Если вы записываете данное ограничение как отдельное выражение, необходимо использовать ключевое слово **FOR colname**.

Рассмотрим пример использования ограничения **Constraint**. В данном примере создается таблица, колонка **accept_date** которой будет принимать значение текущей даты при добавлении новой записи в таблицу, - безусловно, только в том случае, если не будет указано явное значение.

```
CREATE TABLE autostore.yourreadness.child_table
(tree char(20),
branch money,
childs varchar(25),
accept_date datetime DEFAULT getdate()
FOREIGN KEY (tree,branch) REFERENCES pat_table(tree,branch)
```

Ограничение **Check** лимитирует список значений, которые мы можем ввести в колонку. Оно имеет следующий синтаксис:

```
[CONSTRAINT constraintname]
CHECK [NOT FOR REPLICATION] (expression)
```

В случае, если вы используете опцию **NOT FOR REPLICATION**, данное правило проверки уровня поля не будет срабатывать при операции репликации таблицы, в которой оно используется.

Рассмотрим пример использования ограничения **Check**. В следующем примере параметр **Check** ограничивает ввод в колонке **accept_date** значениями, не превышающими дату следующего за текущим дня.

```
CREATE TABLE autostore.yourreadness.child_table
(tree char(20),
branch money,
childs varchar(25),
accept_date datetime CHECK accept_date <= getdate()+1
FOREIGN KEY (tree,branch) REFERENCES pat_table(tree,branch)
```

Создание представлений

Представления служат для вывода информации из одной или нескольких таблиц путем использование запроса. Синтаксис создания представления существенно короче, чем у команды **CREATE TABLE**.

```
CREATE VIEW [owner.]viewname
[(columnname [, columnname]...)]
[WITH ENCRYPTION]
AS selectstatement [WITH CHECK OPTION]
```

Здесь аргумент *owner* определяет владельца этой таблицы. *Viewname* - название представления. Далее, по выбору, можно указать свои собственные названия колонок, перечислив новые наименования в скобках. Если использовать предложение **WITH ENCRYPTION**, то можно затруднить пользователям возможность узнать, откуда представление получает данные. Предложение **WITH CHECK OPTION** позволяет в случае модификации записи посредством представления гарантировать, что запись будет доступна через представление, после того как изменения будут записаны на диск. После ключевого слова **AS** записывается выражение команды **SQL-SELECT**.

Рассмотрим пример создания представления:

```
CREATE VIEW yourreadness.somekindofview
AS SELECT tree FROM Child_table
```

Создание триггеров

Триггеры создаются командой **CREATE TRIGGER**, имеющей следующий синтаксис:

```
CREATE TRIGGER [owner.]trigger_name
ON [owner.]table_name
FOR {INSERT, UPDATE, DELETE}
[WITH ENCRYPTION]
AS sql_statements
```

Есть возможность использовать альтернативный синтаксис, который выглядит так:

```
CREATE TRIGGER [owner.]trigger_name
ON [owner.]table_name
FOR {INSERT, UPDATE}
[WITH ENCRYPTION]
AS
IF UPDATE (column_name)
[{AND | OR} UPDATE (column_name)...] sql_statements
```

Коротко о триггерах можно сказать как об особого рода хранимых процедурах, которые автоматически срабатывают во время операций модификации, удаления или добавления записей. Триггеры используются, как правило, для создания бизнес-правил и поддержки ссылочной целостности данных. Естественно, что непомерное использование триггеров замедляет работу.

Предложение **WITH ENCRYPTION**, так же как и в случае с представлениями, служит для того, чтобы скрыть от постороннего глаза SQL выражение, которое он запускает.

Существуют определенные ограничения на использование триггеров. Нельзя использовать их в представлениях. Рекомендуется не использовать SQL выражения, которые возвращают наборы данных.

8.2. Использование Visual FoxPro для разработки клиентского приложения

Итак, в предыдущем параграфе мы освоили необходимый минимальный набор команд, которого достаточно, чтобы начать работать с данными на SQL Server. Настала пора обсудить вопросы построения клиентской части, при этом вспомнив набор из шести операций, которые нам доступны из приложения front-end. В итоге получается, что из клиентской части нам доступны все операции. На всякий случай не забудьте про возможность недостаточного совершенства имеющегося драйвера ODBC.

В этом параграфе мы обсудим вопросы создания приложения типа клиент-сервер с помощью Visual FoxPro.

Visual FoxPro предоставляет следующие средства для работы с данными, имеющими другой формат:

1. Конструктор соединения (Connection Designer). Альтернативно можно использовать команду **CREATE CONNECTION**. В этом случае для многих установок надо использовать функцию **DBSETPROP()**.
2. Конструктор представления (View Designer). Можно использовать команду **CREATE SQL VIEW**. В этом случае свойства полученного представления необходимо изменять с помощью функции **DBSETPROP()**.
3. Набор функций SQL pass-through позволяет контролировать работу сервера с помощью диалекта SQL самого сервера. Главным образом используется для получения выборки данных для дальнейшей обработки в клиентской части приложения. В то же время вам становятся доступны практически все команды и возможности сервера.

Конструктор соединения используется для создания и модификации соединений, хранимых в базе данных. Активизируется Конструктор соединения, так же как и другие дизайнеры в Visual FoxPro, тремя способами:

1. В меню *File* выполнить команду *New*, затем выбрать *Connection*.
2. Из командного окна с помощью команды **CREATE CONNECTION**.
3. Из *Project Manager*, выбрав *Connection*, *New*.

Соединения хранятся только в базах данных, поэтому необходимо, чтобы во время создания соединения была открыта база данных, лучше та, в которой это соединение будет использоваться.

После запуска вам необходимо последовательно заполнить текстовые поля, которые называются **Data Source**, **UserID** и **Password**. Здесь есть один момент, который необходимо отметить. **Data Source** уже должен быть создан посредством Администратора ODBC. Как уже указывалось, обычно после установки Visual FoxPro, Администратор ODBC уже присутствует на вашем компьютере. Запустите Администратор ODBC и создайте **Data Source** с использованием ODBC драйвера, в нашем случае это SQL Server (рис. 8.2). С помощью кнопки **Drivers** вы можете проверить наличие имеющихся на компьютере драйверов ODBC. Для создания **Data Source** необходимо нажать на кнопку **Add** и в появившемся диалоговом окне заполнить необходимые поля. Имена **Data Source** старайтесь давать осмысленные, чтобы в дальнейшем помнить, какой **Data Source** за что отвечает. Далее заполните имя сервера и в диалоге **Options** укажите имя базы данных, к которой вы будете подсоединяться. В примерах мы будем подсоединяться к базе данных **Pubs**, которая поставляется как образец с MS SQL Server. В зависимости от используемого драйвера диалоговые окна, открывающиеся после нажатия кнопки **Add** и выбора нужного драйвера, несколько различаются, но, как правило, главное, что нам нужно - это имя базы данных, доступ к таблицам которой мы хотим иметь, путь или имя папки, в которой хранятся таблицы, если приложение не работает с базой данных как с контейнером таблиц.

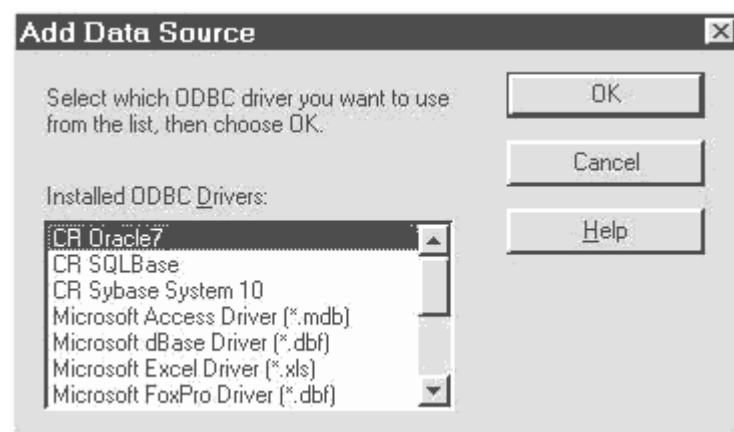


Рис. 8.2. Выбор необходимого ODBC драйвера

Создав **Data Source**, мы имеем все для подготовки соединения и записи его определения в текущую базу данных. Выберите из комбинированного списка с заголовком "**Data Source**" ваш **Data Source** (извините за тавтологию), затем введите ваше пользовательское имя в текстовое поле **UserID** и пароль в текстовое поле **PassWord**. Сверху от трех полей, которые мы заполнили, находятся две кнопки выбора, правая называется "**Connection String**". Если вам больше нравится вводить **Connection String**, то выберите эту кнопку выбора, - в таком случае вы берете на себя не очень обременительную, но все-таки обязанность написать строку соединения самостоятельно, не делегировав эти права Visual FoxPro. Вам нужно набрать что-либо подобное (рис. 8.3):

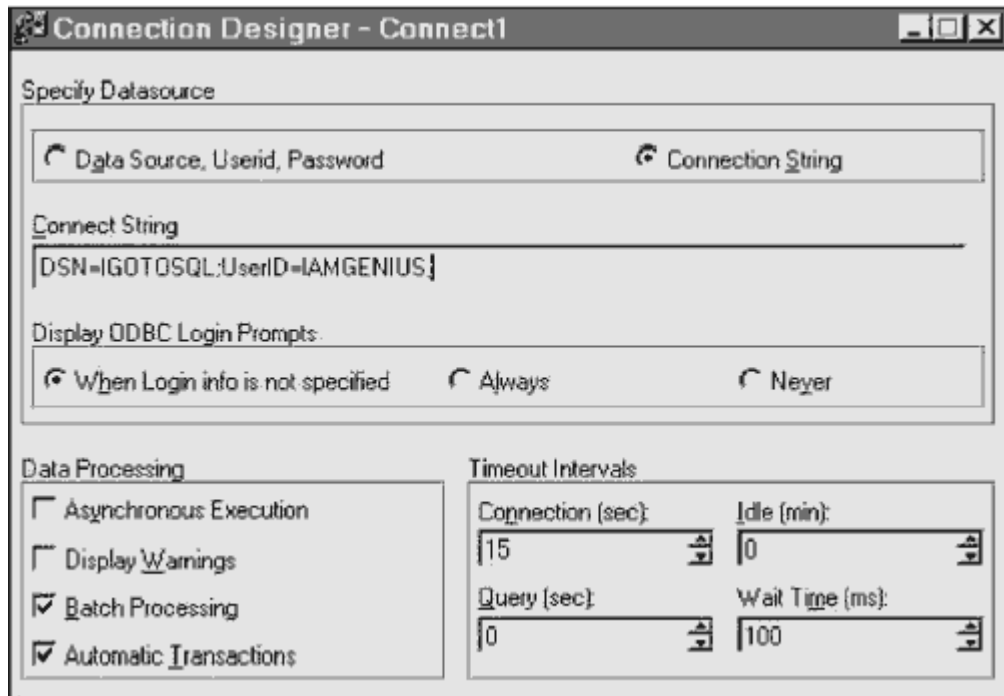


Рис. 8.3.

DSN=IGOTOSQL;UserID=IAMGENIUS;_
 Password=JesusChristSuperStar;Database=Pubs

При этом не используйте никаких кавычек, иначе ODBC вместо установки соединения вернет ошибку. В случае использования строки соединения вы сами выполняете ту работу, которую выполнил бы Visual FoxPro, в том случае, если бы вы ввели всю информацию отдельно в соответствующие текстовые поля.

Далее нужно выбрать, будет ли появляться диалоговое окно с просьбой ввести пользовательское имя при каждом использовании соединения для доступа к таблицам базы данных на сервере. Предлагаются три варианта выбора:

1. Окно будет появляться, если не будет указано пользовательское имя в строке соединения.
2. Окно будет появляться всегда.
3. Окно не будет появляться никогда.

Команда **CREATE CONNECTION** может использоваться для того, чтобы создавать соединения программным путем, правда, при этом если вы не укажете предложение **DATASOURCE**, то все равно на экране появится Конструктор соединения. База данных также должна быть открыта, как и при создании Соединения с помощью Конструктора соединения.

Соединение, созданное в предыдущем примере, программным путем можно создать так:

```
CREATE CONNECTION mysqlconnect ;
DATASOURCE "igotosql" ;
USERID "Iamgenius" PASSWORD "JesusChristSuperStar"
```

Как уже было упомянуто, SQL pass-throught - это средство по созданию SQL команд и передачи их в базу данных SQL. Первой мы рассмотрим функцию **SQLCONNECT()**, которая устанавливает соединение с источником данных. С помощью этой функции можно создать стандартное соединение (Standard Connection), строку соединения (Connect string) и поименованное соединение (Name Connection).

Ниже приводятся примеры для создания каждого типа соединения.

Стандартное соединение

```
hndl=SQLCONNECT('igotosql','iamgenius')
```

Строка соединения

```
hndl=SQLCONNECT('igotosql','iamgenius', 'JesusChristSuperStar','pubs')
```

Поименованное соединение

```
hndl=SQLCONNECT('mysqlconnect')
```

После создания соединения мы можем установить его свойства. Это возможно практически для всех свойств, кроме двух, связанных с внутренними указателями ODBC: **ODBCdbc** и **ODBCstmt**. Свойства можно устанавливать с помощью Конструктора соединения или с помощью функции **SQLSETPROP()**. Для проверки текущих установок свойств используется функция **SQLGETPROP()**.

Функция **SQLSETPROP()** имеет следующий синтаксис:

SQLSETPROP(*nConnectionHandle*, *cSetting*, *eExpression*)

Функция **SQLSETPROP()** устанавливает свойства активного соединения. Надо использовать эту функцию для установки свойств на уровне соединения. Чтобы установить эти свойства на уровне окружения Visual FoxPro и сделать их значениями по умолчанию, нужно использовать вместо указателя соединения (аргумент *nConnectionHandle*) значение 0.

Функция **SQLSETPROP()** возвращает 1, если установка свойства завершилась успешно, -1, если случилась ошибка на уровне соединения, или -2, если ошибка случилась на уровне окружения. Свойство **ConnectTimeout** может быть установлено только на уровне Visual FoxPro. Все остальные свойства можно устанавливать как на уровне Visual FoxPro, так и на уровне соединения. Установки, сделанные на уровне Visual FoxPro, являются значениями по умолчанию для всех последующих вновь создаваемых соединений.

Функция **SQLGETPROP()** служит для чтения свойств текущего соединения. Если аргументом вместо указателя на соединение будет значение 0, то мы получим текущие установки по умолчанию для текущего окружения Visual FoxPro.

Синхронный и асинхронный процессы

Когда вы делаете запрос к внешнему источнику данных, Visual FoxPro выполняет запрос синхронно. (Синхронный процесс - это процесс по умолчанию для Visual FoxPro.) Visual FoxPro не возвращает контроль приложению, пока SQL выражение не выполнится полностью. Программа останавливает свою работу и ждет, пока не обработается весь запрос и данные не вернутся к клиенту. Тем не менее при использовании SQL pass-through вы можете сделать процесс асинхронным, то есть программа может выполнять свои следующие команды, а выборка будет идти в фоновом режиме, заполняя курсор набором данных. Например:

```
myhandle=SQLCONNECT("mysqlconnect")
IA=SQLGETPROP("myhandle", "Asynchronous")
IF !IA=SQLSETPROP("myhandle", "Asynchronous", .T.)
ENDIF
```

Свойство **BatchMode** определяет, как будут выбираться множественные наборы данных, когда используется функция **SQLEXEC()**. Значение по умолчанию для свойства **BatchMode** - истина. При этом не возвращается никаких результатов от функции **SQLEXEC()**, пока функция не завершит свое выполнение.

Например, если функция **SQLEXEC()** содержит два выражения SQL SELECT, никаких результатов не будет возвращено, если оба процесса запущены в режиме **BatchMode**, установленным в истину.

Если режим не пакетный, то результаты возвращаются в паре с функцией **SQLMORERESULTS()**. Например:

```
myhandle=SQLCONNECT("mysqlconnect")
=SQLSETPROP(myhandle, "BatchMode", .T.)
=SQLEXEC(myhandle, "SELECT * FROM account", ;
"SELECT * FROM Country", "Results")
mores=0
DO WHILE mores << 2
    mores=SQLMORERESULTS(myhandle)
ENDDO
```

Сочетание значений двух свойств дает нам четыре комбинации возможных режимов:

Синхронный	Асинхронный
Synchronous Batch	Asynchronous Batch
Synchronous NonBatch	Asynchronous Nonbatch

Синхронный пакетный режим (Synchronous Batch Mode) - выражения, отправленные на сервер, не будут возвращать контроль программе, пока все наборы данных не будут выбраны.

Асинхронный пакетный режим (Asynchronous Batch Mode) - выполнение в данном режиме будет возвращать 0 при каждом обращении функции, породившей процесс, пока не будут возвращены все наборы результатов.

Синхронный не пакетный режим (Synchronous Nonbatch Mode) - выражения, выполняемые в этом режиме, выберут первый набор и вернут 1. Далее должна вызываться функция **SQLMORERESULTS()**, повторяясь, пока не будут выбраны все наборы данных. Если необходимы разные названия для полученных курсоров, новое имя можно указать при обращении к функции **SQLMORERESULTS()**. Функция **SQLMORERESULTS()** вернет 2, когда все наборы данных будут выбраны.

Асинхронный не пакетный режим (Asynchronous NonBatch Mode) - для того чтобы вернуть все наборы результатов, необходимо после того, как функция, вызвавшая процесс, вернет 1, вызывать функцию **SQLMORERESULTS()**, пока она не вернет 2.

Основные свойства соединений перечислены в табл. 8.4.

Таблица 8.4 Основные свойства соединений

Свойство	Описание
Connect-TimeOut	Указывает время ожидания (в секундах), после которого должно возвратиться сообщение об ошибке соединения. Если указать 0, то время ожидания неопределенно и сообщение об ошибке не будет получено никогда. Это свойство может принимать значение от 0 до 600. Значение по умолчанию - 15 секунд. Доступно для чтения и записи.
Idle-Timeout	Указывает промежуток времени, после которого активное соединение деактивируется. При этом соединение не должно использоваться в течение этого промежутка времени. Значение по умолчанию - 0 (интервал бесконечен). Доступно для чтения и записи.
Query-TimeOut	Указывает время ожидания перед возвратом сообщения об ошибке, в случае неудачного выполнения запроса соединения. Если указать значение 0, то сообщение об ошибке никогда не будет возвращено. Может принимать значение от 0 до 600. Доступно для чтения-записи.
Transactions	Содержит числовое значение, которое определяет, как соединение управляет транзакциями внешней таблицы. Установка может принимать следующие значения: 1 или DB_TRANSAUTO (из VISUAL FOXPRO.H) - является значением по умолчанию; транзакции для внешней таблицы обрабатываются автоматически; 2 или DB_TRANSMANUAL (из VISUAL FOXPRO.H) - транзакции обрабатываются вручную через функции SQLCOMMIT() и SQLROLLBACK() .
WaitTime	Время в миллисекундах, задающее интервал, с которым Visual FoxPro проверяет, закончилось ли выполнение операторов SQL. Значение по умолчанию - 100 миллисекунд. Доступно для чтения и записи.

Функция **SQLSTRINGCONNECT()** позволяет устанавливать соединение с помощью строки соединения. Таким образом, вы можете обойтись без хранения определения соединения в базе данных. Параметры строки могут отличаться в зависимости от используемого драйвера. Описание параметров приводится в документации для используемого драйвера. Например:

```
h=SQLSTRINGCONNECT('dsn=MyAutostoreSQL;uid=sa;pwd=Immanager')
=SQLEXP(h,"SELECT * FROM Account","FoxAccount")
```

Если вы создадите соединение, которое будет, к примеру, иметь название C1 и строку соединения, которую мы использовали в первой строчке последнего примера, то можно будет использовать следующую процедуру, итогом которой будет совершенно аналогичный результат.

```
CREATE CONNECTION c1;
CONNSTRING 'dsn=MyAutostoreSQL;uid=sa;pwd=Immanager'
```

```
h=SQLCONNECT('c1')
=SQLEXP(h,"SELECT * FROM Account","FoxAccount")
```

Каждую операцию по соединению с источником данных необходимо завершать отсоединением. В противном случае на каком-то этапе количество активных соединений превысит возможности вашего приложения. Для этого используется функция **SQLDISCONNECT()**. Например:

```
CREATE CONNECTION c1;
CONNSTRING 'dsn=MyAutostoreSQL;uid=sa;pwd=Immanager'
h=SQLCONNECT('c1')
=SQLEXP(h,"SELECT * FROM Account","FoxAccount")
=SQLDISCONNECT(h)
```

При этом следует отметить следующий факт. Получив курсор с помощью функции **SQLDISCONNECT()** и выполнив операцию **SQLDISCONNECT()**, вы не заметите никаких изменений при работе с курсором. Здесь стоит глубже вникнуть в суть процесса. При выполнении функции **SQLDISCONNECT()** мы получаем курсор в памяти, который по большому счету никак не связан с данными на сервере. Здесь мы вплотную соприкасаемся с понятием модифицируемый курсор. Сейчас необходимо вспомнить функцию **CURSORSETPROP()**, о которой мы много говорили в [предыдущей главе](#).

Получив курсор, для установки необходимых вам свойств используйте функцию **CURSORSETPROP()**, чтобы изменения, которые вы проводите над данными в курсоре, отображались на сервере.

Если в следующей процедуре отключить связь до закрытия курсора, удалив строчку с единственной командой **USE**, все равно нельзя будет в дальнейшем проводить обновления в этом наборе данных.

```
CREATE CONNECTION c1;
CONNSTRING 'dsn=MyAutostoreSQL;uid=sa;pwd=Immanager'
h=SQLCONNECT('c1')
h=SQLCONNECT("cn1")
=SQLEXP(h,"SELECT * FROM Account","FoxAccount")
=CURSORSETPROP('tables','Account')
=CURSORSETPROP('KeyFieldList','field1')
=CURSORSETPROP('updatablefieldlist','field2')
=CURSORSETPROP('updatenamelist',;
'field1 mytable.field1,field2 mytable.field2')
=CURSORSETPROP('sendupdates',.T.)
BROWSE
USE
=SQLDISCONNECT(h)
```

Курсор, полученный с помощью функции **SQLDISCONNECT()**, имеет оптимистическую буферизацию записи. С помощью функции **CURSORGETPROP()** вы можете установить нужный вам тип буферизации. Помимо этого следует обратить внимание на тип обработки транзакций. Если свойство **Transactions** равно 1, то у вас нет практически никакого контроля над транзакциями, так как они являются автоматическими. Если установить это свойство равным 2, то вы сможете управлять процессом транзакций. Например, при изменении содержимого нескольких записей, в зависимости от сложившихся обстоятельств, вы сможете либо отменить все изменения, либо записать их на диск.

Для этого используются функции **SQLCOMMIT()** и **SQLROLLBACK()**. Параметром обеих функций является указатель вашего соединения. При этом учтите, что если вы не используете эти функции и просто закрываете курсор, то изменения автоматически запишутся на диск, естественно, если они не вступают в конфликт с какими-либо условиями или обстоятельствами, обычно появляющимися при работе в многопользовательском режиме.

В следующем примере в зависимости от дня недели изменения либо записываются на диск, либо происходит откат. При этом предварительно устанавливается требуемое значение свойства **Transactions** для активного соединения.

```
CREATE CONNECTION cn1 CONNSTRING 'dsn=myteach'
h=SQLCONNECT("cn1")
=SQLEXP(h,"SELECT * FROM Account","FoxAccount")
=SQLSETPROP(h,"Transactions",2)
=CURSORSETPROP('tables','Account')
```

```

=CURSORSETPROP('KeyFieldList','account,key_customer')
=CURSORSETPROP('updatablefieldlist','summa')
=CURSORSETPROP('updatenamelist',;
'account account.account,key_customer account ; key_customer,;
summa account.summa')
=CURSORSETPROP('sendupdates',.T.)
REPLACE ALL field2 WITH 27
IF DAY(DATE())=2
    SQLROLLBACK(h)
ELSE
    SQLCOMMIT(h)
ENDIF
USE
=SQLDISCONNECT(h)

```

Создание внешних представлений

В [главе 7](#) обсуждались представления и работа с ними. Далее мы продолжим разговор о них, но будем работать с данными либо внешнего формата, либо с данными смешанного характера, то есть исходные таблицы могут быть как формата Visual FoxPro, так и любого внешнего формата. Представьте, например, ситуацию такого рода. Ваша организация работала с данными с помощью приложений, написанных с использованием FoxPro 2.6 или более ранних версий. Но вот принято решение о переходе на версию Visual FoxPro 3.0 или выше. Можно поступить следующим образом. Ждать, пока программисты переписут все приложения с использованием новой версии, потом, когда новая версия будет готова, отладить ее, подготовить соответствующее аппаратное обеспечение и начать работать. А можно поступить по-другому. Перейти частично на новую версию, а файлы данных хранить в формате FoxPro 2.6, там где их еще используют старые приложения, постепенно переводя их в новый формат по мере обновления техники и перевода отдельных частей приложений на новую версию.

Рассмотрим синтаксис команды **CREATE SQL VIEW** для создания внешнего представления.

```

CREATE SQL VIEW [ViewName] [REMOTE]
[CONNECTION ConnectionName] [SHARE]
| CONNECTION DataSourceName]
[AS SQLSELECTStatement]

```

Для создания внешнего представления надо указывать ключевое слово **REMOTE**. Далее вы можете с помощью ключевого слова **CONNECTION** создать представление либо с помощью соединения, либо с помощью источника данных. Таким образом, во втором случае вы минуete соединение. Хорошо это или плохо? Это надо решать для конкретной задачи индивидуально. В случае использования соединения вы получаете более полный контроль над процессом получения данных, так как у вас, исходя из вышеизложенного, есть функции, с помощью которых вы легко можете управлять свойствами соединения. Представление помимо этого может иметь соединение, разделенное между несколькими представлениями. Таким образом, меняя свойства одного соединения, вы получаете контроль над несколькими наборами внешних данных.

Перед тем как вы начнете создавать внешнее представление, лучше уже иметь готовое определение соединения в текущей базе данных или подходящий набор источников данных, полученных с помощью ODBC, хотя это и не является необходимым требованием. Поэтому выберите один из способов построения внешних запросов. Если вы работаете с использованием Project Manager, то выберите нужную базу данных и, установив курсор на пункте Remote Views, как это показано на рис. 8.4, нажмите кнопку New.

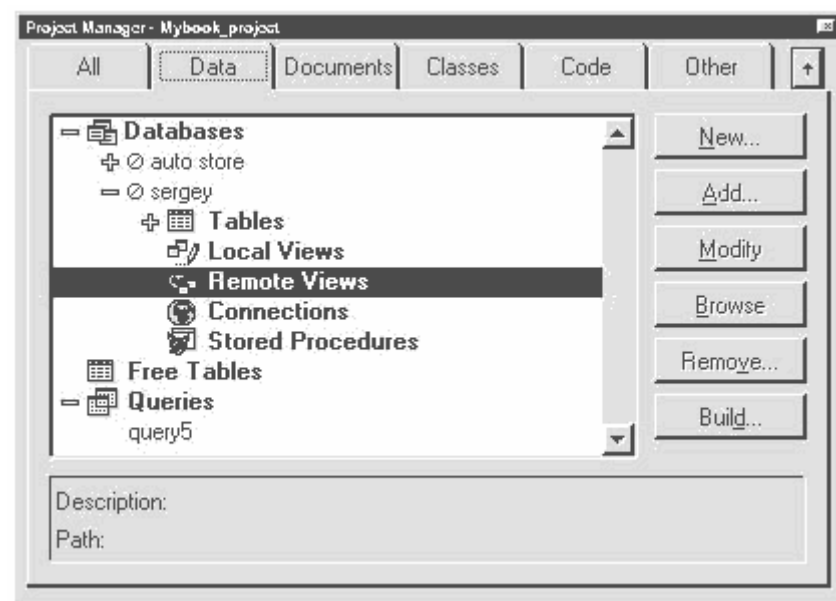


Рис. 8.4.

После этого появится диалоговое окно, в котором вам будет предложен выбор среди имеющихся в текущей базе данных соединений или среди источников данных, установленных на вашем компьютере. При этом вы можете создать новое соединение и затем снова вернуться в текущее диалоговое окно. С источниками данных дело обстоит хуже, но в конце концов мы работаем в многозадачной среде. Загрузите ODBC, создайте нужный источник данных, естественно при наличии на компьютере необходимого драйвера, и снова переключитесь в Visual FoxPro.

В зависимости от того, к данным какого формата вы хотите обратиться, будут доступны либо таблицы из конкретного каталога, либо из конкретной базы данных. В случае с SQL Server и Microsoft Access это будут таблицы из баз данных, которые вы укажете в источнике данных.

После того как вы выберете необходимое вам соединение или источник данных и нажмете кнопку ОК, станет доступен Конструктор представлений вместе с диалогом по выбору таблиц из внешней базы данных, к которой вы присоединились. В остальном различий с Конструктором локальных представлений нет.

Если необходимо смешивать локальные данные с внешними, то надо построить по крайней мере два представления. Первое - это представление, которое будет состоять из данных на сервере, а второе будет состоять из данных из локальной таблицы или таблиц и полученного на первом этапе представления. Для того что ваши изменения после модификации данных в полученном представлении второго уровня отображались на сервере, необходимо использовать функцию **TABLEUPDATE()** дважды - вначале в представлении второго уровня, затем в представлении первого уровня. Другой способ - это последовательно закрыть представление второго уровня, а затем первого.

Выше были описаны два способа работы с внешними данными, **SQL pass-through** и внешние представления. **SQL pass-through** требует большего объема программирования, но в то же время, используя эту технологию, вы практически не ограничены в своих возможностях по управлению внешними данными. Внешние представления легче использовать, значительно меньше объем программирования, но с их помощью можно только модифицировать данные: редактировать, удалять и добавлять записи.

Любой набор данных или курсор, который вы получите, можно просматривать с помощью окна **Browse** или объекта **Grid**. Говоря другими словами, вся эта технология тесно интегрирована. Объекты **Grid** или **Browse** сами по себе можно считать средствами управления данными, по гибкости не имеющими аналогов среди других популярных систем. Можете проверить, что скорость доступа к данным уменьшается незначительно, даже если эти объекты активны. Все замедление отнесите на счет вашего видеоадаптера, а объекты достаточно быстро отображают искомую информацию.

Но все-таки у вас могут возникнуть потребности выбирать данные еще быстрее, по крайней мере будет казаться, что вы можете этого добиться, если залезете в глубь процесса технологии ODBC или какой-либо другой, в случае если ее функции будут доступны с помощью какого-нибудь API. Тогда вспомните, что Visual FoxPro поддерживает вызовы DLL, что, впрочем, делали и его недавние предшественники, правда, посредством немного более сложных манипуляций.

Поэтому в следующем разделе при обсуждении вопросов взаимодействия Access и Visual Basic с внешними данными все, что касается доступа к данным через ODBC API, RDO (объекты доступа к внешним данным), SQL-DMO, в равной мере относится и к Visual FoxPro.

8.3. Использование Access и Visual Basic для разработки клиентского приложения

Access и Visual Basic предоставляют разнообразные средства для работы с внешними данными.

В этом параграфе мы кратко расскажем об этих средствах, снабдив изложение небольшими примерами, которые вы можете развить до более высокой степени функциональности.

Технологии, доступные как в обоих продуктах, так и в каждом по отдельности, будут описываться в зависимости от степени сложности, а не отдельно для каждого пакета. Это делается специально, так как переход от одной среды разработки к другой достаточно легок в связи с тем, что в основе этих продуктов лежит один язык.

При работе в Microsoft Access самый простой способ обработки информации с SQL Server - использование присоединенных таблиц. Для этого используйте команду *Внешние данные* из меню *Файл*. Из двух опций *Импорт* и *Связь с таблицами* нас интересует вторая, так как после импорта способы передачи обновленной информации на сервер не так легки, как во втором случае. После того как вы выберете команду *Связь с таблицами*, вы попадете в диалоговое окно *Связь*. Главный элемент на первом этапе для нас - это список *Тип файла*. В этом списке перечисляются все типы файлов, с которыми работает процессор баз данных Microsoft Access, достаточно впечатляющий по количеству поддерживаемых СУБД различных версий. Но нас интересует элемент, который расширяет этот список еще значительно. Как вы уже догадались, этот элемент называется *базы данных ODBC* (рис. 8.5).

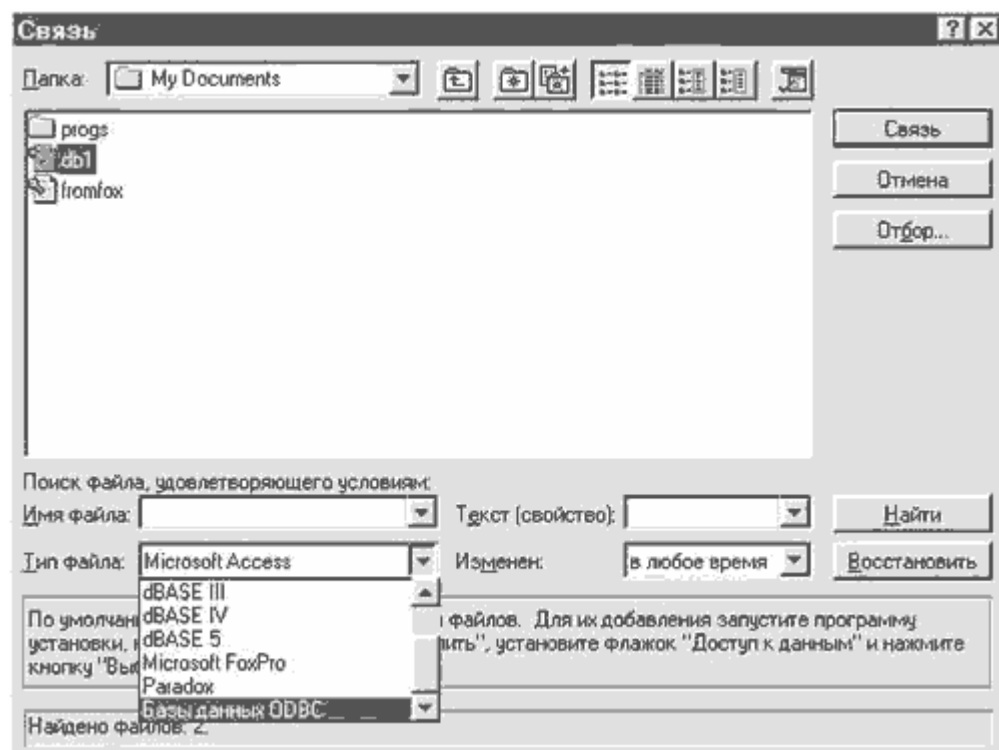


Рис. 8.5. Диалоговое окно *Связь* в Access 7.0

После выбора этого элемента перед вами возникнет следующее диалоговое окно, в котором вы увидите список всех источников данных (*Data Source*), доступных на компьютере. Если нужного источника данных в списке нет, то вы с помощью кнопки *New* легко можете перейти в Администратор ODBC и создать требуемый источник (рис. 8.6).

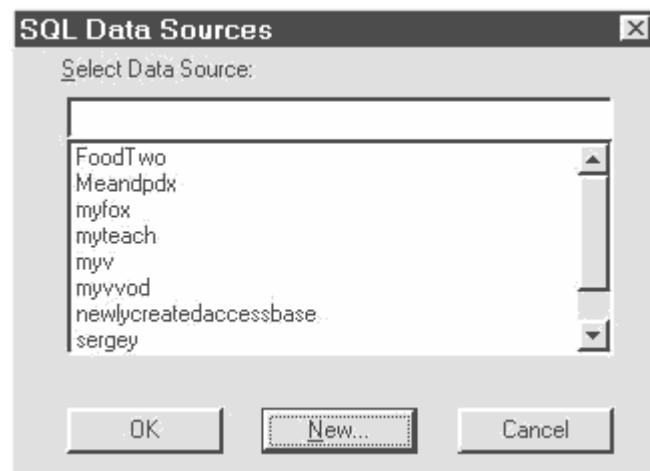


Рис. 8.6.

После выбора источника данных в следующем диалоговом окне вам останется только выбрать из списка доступных таблиц те, которые вы хотите присоединить. При этом обратите внимание, что вы можете выбрать несколько или даже все имеющиеся таблицы. На этом процесс присоединения закончен. В случае отсутствия в присоединяемой таблице уникального индекса вам будет предложено выбрать это поле самостоятельно. Это делается для того, чтобы процессор баз данных Jet Access мог однозначно передавать изменения в данных на сервер. Поэтому тщательно следите, чтобы поле, определяющее уникальность записи, всегда присутствовало в ваших таблицах. Теперь вы можете работать с внешней таблицей так же, как и с таблицей самого Access. Вам будут недоступны только операции по модификации структур таблиц непосредственно визуальными средствами Access. Обратите внимание, что в предыдущем предложении сказано "непосредственно визуальными средствами". Потому что способы изменить структуру таблиц на сервере у вас еще остаются.

Среди дополнений (Add-In) Visual Basic присутствует Data Manager. Если на компьютере установлен Visual Basic, то в качестве упражнения предлагаем проделать аналогичную манипуляцию с присоединением внешних таблиц к базе данных с помощью Data Manager. В случае затруднений обратите внимание на рис. 8.7-8.9.

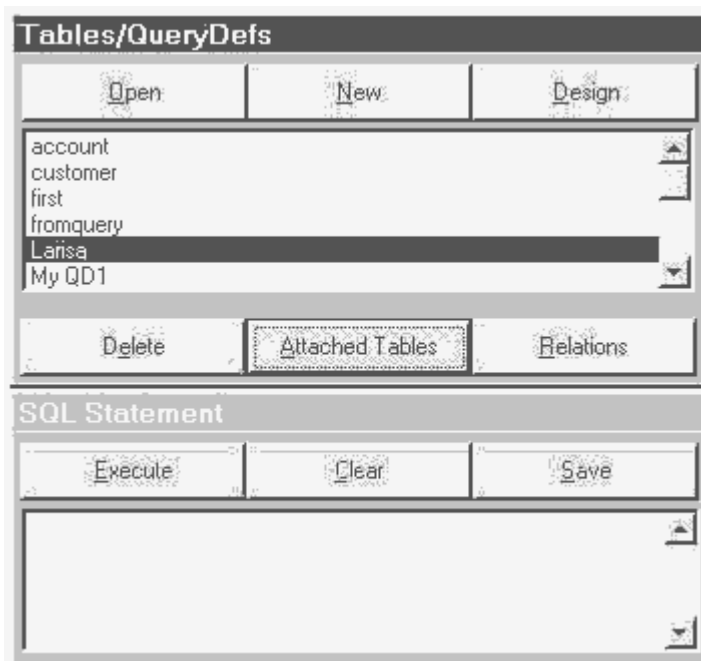


Рис. 8.7.

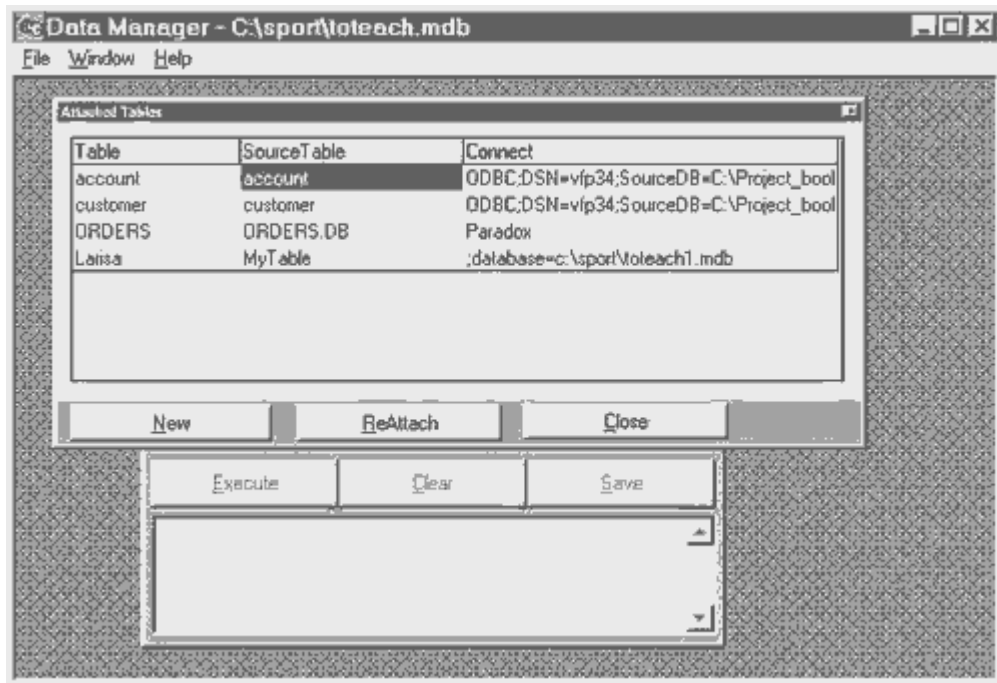


Рис. 8.8.

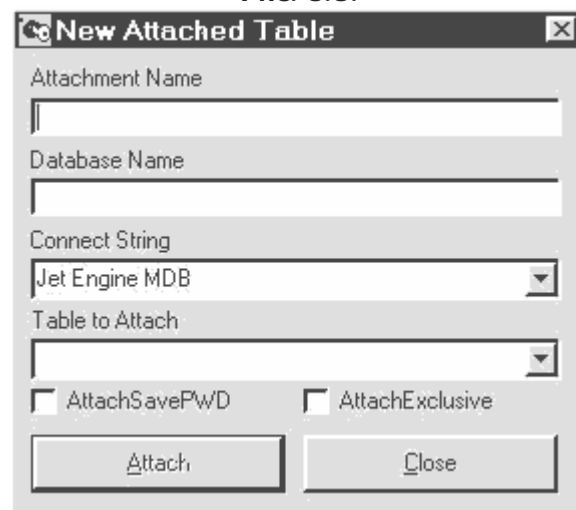


Рис. 8.9.

Чем хороши присоединенные таблицы? Во-первых, не требуется никакого программирования для того, чтобы работать с ними. Во-вторых, они модифицируемы, то есть все изменения, которые вы будете проводить с записями, будут отражаться на сервере. Исключением является случай, когда Access однозначно не сможет определить запись на сервере, в которой надо провести обновление данных в колонках.

Чем плохи присоединенные таблицы? Только тем, что в некоторых случаях вам будет не хватать скорости обработки. Какой есть выход из положения? Использовать другие методы для получения набора данных.

Но сначала закончим с присоединенными таблицами. Помимо визуального способа, есть и программный. Для этого мы должны использовать DAO - набор объектов для работы с данными. Вы уже не раз встречались на страницах этой книги с конструкцией следующего типа:

```
Dim MyDb As Database
Dim mytableDef As TableDef
Set MyDb = DBEngine.Workspaces(0).Databases(0)
Set MytableDef = _ CreateTableDef("JustProgrammaticallyMadeTable")
```

Данная конструкция создает таблицу в текущей базе данных. Отметим, что этот пример не завершает построения таблицы, так как мы не создали полей и не добавили таблицу в коллекцию таблиц текущей базы данных.

Следующий пример создает присоединенную таблицу формата Visual Fox-Pro 3.0. Необходимо

лишь добавить две строки, которые устанавливают свойства `Connect` и `SourceTableName` для вновь создаваемого объекта. В данном примере создается присоединенная таблица `AttachTable`. Для этого используется источник данных `vfp34`. На вашем компьютере название таблицы и источника данных (`Data Source`) могут быть иными.

```
Public Sub TableCREATE()
    Dim myDb As DATABASE, mytdef As TableDef
    Set myDb = DBEngine.Workspaces(0)("Autostore.mdb")
    Set mytdef = myDb.CreateTableDef("AttachTable")
    mytdef.Connect = "ODBC;dsn=vfp34"
    mytdef.SourceTableName = "country"
    myDb.TableDefs.Append mytdef
End Sub
```

Данную процедуру можно запустить как в Access, так и в Visual Basic или Microsoft Excel. Единственное отличие последних в том, что необходимо проследить за доступностью объектов DAO приложению. После выполнения этого или подобного кода в базе данных появится значок, а если вы откроете присоединенную таблицу в режиме Конструктора, то, воспользовавшись командой *Свойства* меню *Вид*, сможете прочитать в строке Описание строку Соединения.

Не забывайте, что физически присоединенные таблицы не находятся в вашей базе данных, и требуется определенное время для связи с ней, поэтому при работе с таблицами, хранящими большое количество данных, используйте другие, более быстрые методы выборки данных, к примеру, параметрические запросы.

Обратим внимание, что с присоединенными таблицами нельзя использовать метод `Seek`. Поэтому здесь нужно придумать другие способы оптимизации поиска, самым лучшим из которых является, извините за назойливость, параметрический запрос.

Перед тем как использовать транзакции, необходимо выяснить, поддерживаются ли операции такого рода в источнике данных. Для баз данных ODBC нельзя использовать вложенные транзакции.

До сих пор мы говорили о присоединенной таблице, но можно и напрямую открыть таблицу. Это доступно только с помощью языка программирования Basic.

```
Dim CurrentDatabase As Database
Dim MySet As Recordset
'Открываем внешнюю базу данных формата FoxPro
Set CurrentDatabase = DBEngine.Workspaces(0).OpenDatabase_
("C:\FOXPRO\DATA\", False, False, "FoxPro 2.5")
'Открываем таблицу Customer
Set MySet = CurrentDatabase.OpenRecordset("Customer")
```

В приведенном примере в начале делается текущей база данных внешнего формата (в нашем случае, так как FoxPro 2.x не поддерживает понятие базы данных как контейнера таблиц, это каталог `C:\FOXPRO\DATA\`), а затем напрямую открывается таблица покупателей в этом каталоге. Понятно, что никакого значка для данной таблицы в текущей базе данных не появится и работать с этими данными мы сможем только программным способом. Еще одним недостатком данного метода доступа к внешним данным является более медленная, по сравнению с присоединенной таблицей, скорость работы.

Помимо присоединения уже существующих таблиц, хранящихся в источнике данных, вы можете создавать новые таблицы, которые станут присоединенными.

Ниже приводится простейший пример создания таблицы формата FoxPro 2.6.

```
Dim CurrentDatabase As Database
Dim MyTableDef As TableDef
Set CurrentDatabase = DBEngine.Workspaces(0).OpenDatabase("C:\DATA", False, False, "FoxPro 2.6")
Set MyTableDef = CurrentDatabase.CreateTableDef("FromAccess")
MyTableDef.Fields.Append MyTableDef.CreateField("Field1", DB_TEXT, 15)
CurrentDatabase.TableDefs.Append MyTableDef
```

Следующим способом для работы с данными на сервере, который мы рассмотрим, будет `SQL pass-through`. Выборка данных при использования `SQL pass-through` проходит быстрее, так как запрос сразу отправляется на сервер, минуя процессор баз данных Jet. Но полученные запросы являются не модифицируемыми, то есть изменения в них не передаются на сервер. Следовательно, вам нужно использовать какие-то другие способы для обновления данных на

сервере, например, использовать запросы модификации. Важным аргументом в пользу использования SQL *pass-through* запросов является, как уже было сказано выше, более высокая скорость выполнения, но, помимо этого, с помощью SQL *pass-through* вам доступны не только команды выборки, но и команды определения данных, возможность запускать хранимые процедуры. В итоге вы получаете, в зависимости от данных вам прав, довольно значительные возможности по управлению данными на сервере. По большому счету, визуальнo запросы *pass-through* не создаются, но какие-то элементы автоматизации присутствуют. Для того чтобы создать запрос с помощью Конструктора Запросов, перейдите на вкладку Запросы в контейнере базы данных. Затем в меню *Запрос* выберите команду *Запрос SQL* и опцию *К серверу*, которая в английской версии называется *Pass-Through*. После выбора этого пункта меню больше не будет доступен режим конструктора, то есть SQL команды придется набирать вручную. Не будет доступен даже Построитель. Единственное, что вы можете выбрать - это свойства вашего запроса, среди которых имя источника данных (*Data Source*). Чтобы получить доступ к диалогу свойств, выберите команду *Свойства* в меню *Вид* (рис. 8.10).

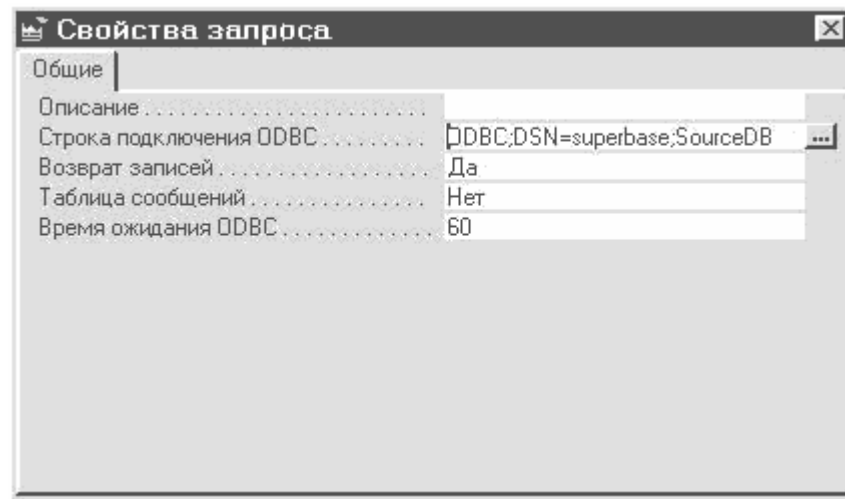


Рис. 8.10.

Среди других свойств следует обратить внимание на Возврат записей. Если вы собираетесь использовать запросы определения данных, то установите значение этого свойства равным "Нет".

В Visual Basic визуальное создание *pass-through* запросов не доступно при работе с Data Manager.

Для того чтобы создавать запросы *pass-through* программно, используются объекты *QueryDef*.

Далее будет разобран небольшой пример, созданный с помощью Visual Basic и использующий *pass-through* запросы для работы с внешними данными. Приложение состоит из одной формы, которая представлена на рис. 8.11.

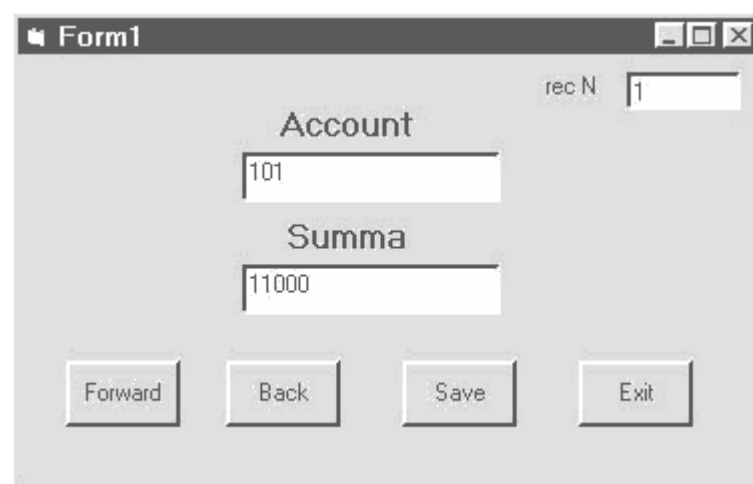


Рис. 8.11.

Данная форма при загрузке использует SQL *pass-through* запрос для создания набора данных, который мы можем просматривать и редактировать.

```

Private Sub Form_Load()
    Set Db = Workspaces(0).OpenDatabase("C:\SPORT\TOTEACH.MDB")
    'Проверка наличия в базе данных
    'запроса с именем "My QD1", и если запрос присутствует, то
    'удаляем его.
    For I = 0 To Db.QueryDefs.Count - 1
        If Db.QueryDefs(I).Name = "My QD1" Then
            Db.QueryDefs.Delete "My QD1"
        Exit For
    End If
Next I
' Мы создаем запрос или, на языке, DAO объект QueryDef и
' устанавливаем строку соединения.
Set Qd1 = Db.CreateQueryDef("My QD1")
Qd1.Connect = "odbc;dsn=vfp34;"
' Записываем SQL запрос и устанавливаем свойство
' ReturnsRecord (Возврат записей) равным Истине, так как
' наш запрос должен возвращать записи
Qd1.SQL = "SELECT * FROM Account"
Qd1.ReturnsRecords = True
' В этой части процедуры мы устанавливаем соответствие
' между полями набора данных, предварительно создав его,
' и объектами формы, а именно двумя текстовыми полями
Set Rs = Db.OpenRecordset("My Qd1",_dbOpenSnapshot)
Me.Text1 = Rs!account
Me.Text2 = Rs!summa
vtext1 = Rs!account
vtext2 = Rs!summa
Me.Text3 = Rs.RecordCount
End Sub

```

Как видно из примера, основное отличие **pass-through** запросов от остальных запросов - это наличие непустого свойства **Connect** и установка свойства **ReturnRecords**.

Если вы отредактируете какое-либо из полей формы, то сможете записать изменения на сервер с помощью **pass-through** запроса. Такой сложный путь необходим ввиду того, что выборки, получаемые в результате выполнения **pass-through** запросов, не модифицируемы.

```

Private Sub Command3_Click()
    For I = 0 To Db.QueryDefs.Count - 1
        If Db.QueryDefs(I).Name = "My QD2" Then
            Db.QueryDefs.Delete "My QD2"
        Exit For
    End If
Next I
Set Qd2 = Db.CreateQueryDef("My QD2")
Qd2.Connect = "odbc;dsn=vfp34;"
Qd2.SQL = "Update account set account=" & Str(Me.Text1) & ",_summa = " & Str(Me.Text2) &
" where account =" & Str(vtext1) & " _and summa = " & Str(vtext2)
Qd2.ReturnsRecords = False
Qd2.Execute
Set Rs = Db.OpenRecordset("My Qd1",_dbOpenSnapshot)
Rs.FindNext "account=" & Str(Me.Text1) & " _and summa=" & Str(Me.Text2)
Me.Text3 = Rs.RecordCount
End Sub

```

Обратите внимание, что свойство **ReturnsRecords** устанавливается равным **False** и на обращение к методу **Execute** объекта **QueryDef**.

Но основным предназначением использования **pass-through** запросов все же считается не выборка данных, а запросы определения данных и выполнение хранимых процедур на сервере.

В следующем примере создается таблица на внешнем по отношению к Access источнике данных (**Data Source**)

```

Set Qd3= Db.CreateQueryDef("My QD3")
Qd3.Connect = "odbc;dsn=vfp34;"
Qd3.SQL = "CREATE TABLE madebyaccess(field1 c(20))

```

Qd3.ReturnsRecords = False
Qd3.Execute

8.4. Использование ODBC API для доступа к внешним данным

ODBC API - прикладной интерфейс программиста для доступа к функциям ODBC. Структура ODBC была изложена в начале этой главы.

В этом параграфе мы познакомимся с некоторыми функциями ODBC более подробно и рассмотрим примеры использования этих функций для доступа к данным на сервере.

Кодирование с помощью ODBC значительно сложнее, но выигрыш в скорости может быть существенным. Здесь следует отметить, что наибольший выигрыш в производительности достигается при доступе к базам данным ODBC. Если же вы попытаетесь с помощью данного метода улучшить показатели доступа к данным, к которым Access обращается с помощью ISAM - последовательного индексного доступа к данным, то результат будет не столь хорош, как хотелось бы, а иногда и совсем плох.

Для того чтобы воспользоваться функциями ODBC API, их необходимо подключить с помощью команды **Declare**.

```
' ODBC API объявления
Declare Function oSQLAllocEnv Lib "odbc32.dll" _
Alias "SQLAllocEnv" (phenv As Long) As Integer
Declare Function oSQLAllocConnect Lib "odbc32.dll" _
Alias "SQLAllocConnect" (ByVal henv As Long, phdbc As _ Long) As Integer
Declare Function oSQLConnect Lib "odbc32.dll" _
Alias "SQLConnect" (ByVal hdbc As Long, ByVal szDSN As _ String, _
ByVal cbDSN As Integer, ByVal szUID As String, _
ByVal cbUID As Integer, ByVal szAuthStr As String, _
ByVal cbAuthStr As Integer) As Integer
Declare Function oSQLAllocStmt Lib "odbc32.dll" _
Alias "SQLAllocStmt" (ByVal hdbc As Long, pHstmt As _ Long) As Integer
```

Четыре функции, которые приведены выше, служат для установки указателя для окружения, соединения, утверждения и установления связи с приложением - сервером. Установка указателя с перечисленными объектами - необходимая процедура для дальнейших действий, которые вы планируете совершить, используя соединение, построенное с помощью ODBC API. Причем вызов этих функций должен происходить в том порядке, в каком они перечислены выше. Далее приводится пример использования этих функций для построения процедуры, которая соединит вас с источником данных. Причем вы не связаны рамками Access или Visual Basic, подойдет любой продукт, который поддерживает доступ к функциям, хранящимся в динамически подключаемых библиотеках DLL.

Перед использованием этой функции необходимо определить структуру следующего типа (в языках, которые не могут создавать типов, придумайте что-нибудь другое, выход всегда есть):

```
Type ORecordSet
    IngHenv As Long ' Указатель окружения
    IngHdbc As Long ' Указатель соединения
    IngHstmt As Long ' Указатель утверждения
    IngHstmtUpdate As Long ' Указатель используемый для
        ' изменений и удалений
    intColumns As Integer ' Возвращаемые колонки
    IngRows As Long ' Возвращаемые записи
    IngRowCount As Long ' Число записей для операций
        ' со многими записями
    IngRowSetSize As Long ' Размер набора записей
        ' в курсоре
    IngMaxRows As Long ' Максимальное число
        ' возвращаемых записей
    IngMaxWidth As Long ' Максимальная ширина поля
    IngCursor As Long ' Тип курсора
    IngConcur As Long ' Тип совпадения
    EOF As Boolean ' Указывает конец файла
    BOF As Boolean ' Указывает начало файла
    IngFirstRow As Long ' Первая запись в курсоре
```

```

lngCurrentRow As Long ' Текущая запись в курсоре
lngLastRow As Long ' Последняя запись в курсоре
lngLastSetRow As Long ' Последняя запись в наборе
    ' данных
strCursorName As String ' Имя курсора
intArrayPos As Integer ' Позиция этого типа
    ' в массиве
intRowStatus() As Integer ' Расширенная информация
    ' о наборе записей
strColLabels() As String ' Текст метки колонки
lngColLabels() As Long ' Длина метки колонки
lngDisplaySize() As Long ' Ширина вывода колонки
End Type

```

Информация о наборе данных или создание переменной типа, который объявлен выше - oRecordSet:

```

Private mtypODBC As OrecordSet
    Помимо этого понадобятся следующие константы:
' Возможные значения, которые будет возвращать вызов ODBC
Public Const SQL_INVALID_HANDLE = -2
Public Const SQL_ERROR = -1
Public Const SQL_SUCCESS = 0
Public Const SQL_SUCCESS_WITH_INFO = 1
Public Const SQL_STILL_EXECUTING = 2
Public Const SQL_NEED_DATA = 99
Public Const SQL_NO_DATA_FOUND = 100

```

Данную функцию необходимо использовать, передавая ей в качестве параметров имя источника данных, идентификатор пользователя и пароля пользователя. Функция должна возвращать ложь или истину.

```

Function rODBCConnect(strServer As String, strUID As String,
    _strPassword As String) As Integer
    Dim intRet As Integer
    Dim strConnOut As String * 256
    Dim intConnOut As Integer
    Dim typOBad As ORecordSet
    Dim i As Integer
    On Error GoTo rODBCConnectErr
    rODBCConnect = True
    ' Установка указателя окружения
    intRet = oSQLAllocEnv(mtypODBC.lngHenv)
    If intRet <<>> SQL_SUCCESS Then
        Call rODBCErrorInfo(mtypODBC.lngHenv, SQL_NULL_HDBC, _
            SQL_NULL_HSTMT)
    End If
    If intRet << SQL_SUCCESS Then
        rODBCConnect = False
        GoTo rODBCConnectExit
    End If
    ' Установка указателя соединения
    intRet = oSQLAllocConnect(ByVal mtypODBC.lngHenv, _
        mtypODBC.lngHdbc)
    If intRet <<>> SQL_SUCCESS Then
        Call rODBCErrorInfo(mtypODBC.lngHenv, _
            mtypODBC.lngHdbc, SQL_NULL_HSTMT)
    End If
    If intRet << SQL_SUCCESS Then
        rODBCConnect = False
        GoTo rODBCConnectExit
    End If
    ' Соединения с указанным драйвером
    intRet = oSQLConnect(ByVal mtypODBC.lngHdbc, strServer, _
        Len(strServer), strUID, Len(strUID), strPassword, _

```

```

        Len(strPassword))
    If intRet <>> SQL_SUCCESS Then
        Call rODBCErrorInfo(mtypODBC.IngHenv, _
            mtypODBC.IngHdbc, SQL_NULL_HSTMT)
    End If
    If intRet << SQL_SUCCESS Then
        rODBCConnect = False
        GoTo rODBCConnectExit
    End If
    rODBCConnectExit:
    Exit Function
rODBCConnectErr:
    MsgBox Err.Number & ": " & Err.Description, _
        vbCritical, "rODBCConnect()"
    typOBad.intArrayPos = -1
    rODBCConnect = False
    Resume rODBCConnectExit
End Function

```

После того как вы установили соединение, вам необходимо установить указатель для утверждения, перед тем как приложение сможет выполнить SQL запрос, как показано в следующем примере:

```

intRet = oSQLAllocStmt(ByVal mtypODBC.IngHdbc,_
    mtypODBC.IngHstmt)

```

К счастью, эту функцию надо выполнять только один раз, так как возвращаемый указатель может использоваться любым количеством соединений. Однако с некоторыми серверами, такими как Microsoft SQL Server, вы должны завершить запрос с указателем утверждением, перед тем как начать выполнение другого запроса. В силу этого, вы не можете разместить указатель другого утверждения без открытия другого соединения к серверу или же вам необходимо подождать, пока текущее выражение не завершит работу. В зависимости от драйвера вы можете иметь возможность выполнять несколько выражений на одном соединении. Используйте функцию **SQLGetInfo()** для того, чтобы определить, способен ли ваш драйвер выполнять несколько утверждений на одном соединении.

Таким образом, мы подошли к процессу выполнения SQL выражения. Для этого мы используем объявление следующей функции:

```

Declare Function oSQLExecDirect Lib "odbc32.dll" _
Alias "SQLExecDirect" (ByVal hstmt As Long, _
    ByVal szSqlStr As String, ByVal cbSqlStr As Integer) As_ Integer

```

Теперь мы объявили все функции для открытия доступа к набору данных, хранящихся на сервере. Эта функция может выглядеть следующим образом:

```

Function rODBCOpenRecordset( _strSQL As String, lngType As Long) As Boolean
' Открывает набор данных на источнике данных ODBC
' Вначале необходимо вызвать ODBCConnect для установки
' соединения
' Возвращает true при успешном завершении, false при
' любой ошибке
    Dim intRet As Integer
    Dim i As Integer
    On Error GoTo rODBCOpenRecordsetErr
    rODBCOpenRecordset = True
    ' Проверка наличия соединения
    If mtypODBC.IngHdbc = SQL_NULL_HDBC Then
        MsgBox "Нет открытых дескрипторов. Нет открытых соединений для открытия наборов
            данных. " & _
            "Вызовите вначале rODBCOpenRecordset", vbCritical, _
            "rODBCOpenRecordset()"
        rODBCOpenRecordset = False
        GoTo rODBCOpenRecordsetExit
    End If

```

```

' Размещает SQL выражение для использования
' в других функциях
intRet = oSQLAllocStmt(ByVal mtypODBC.IngHdbc, _
    mtypODBC.IngHstmt)
If intRet <<>> SQL_SUCCESS Then
    Call rODBCErrorInfo(mtypODBC.IngHenv, _
        mtypODBC.IngHdbc, mtypODBC.IngHstmt)
End If
If intRet << SQL_SUCCESS Then
    rODBCOpenRecordset = False
    GoTo rODBCOpenRecordsetExit
End If
' Устанавливает некоторые значения в глобальной
' структуре
mtypODBC.BOF = False
mtypODBC.EOF = False
mtypODBC.IngFirstRow = 0
mtypODBC.IngCurrentRow = 0
mtypODBC.IngLastRow = 0
mtypODBC.IngCursor = SQL_CURSOR_DYNAMIC
If IngType = dbOpenDynaset Then
    mtypODBC.IngConcur = SQL_CONCUR_VALUES
ElseIf IngType = dbOpenSnapshot Then
    mtypODBC.IngConcur = SQL_CONCUR_READ_ONLY
Else
    "rODBCOpenRecordset()"
    rODBCOpenRecordset = False
    GoTo rODBCOpenRecordsetExit
End If
mtypODBC.IngRowsetSize = odbcRowsetSize
'Устанавливает максимальное количество записей
mtypODBC.IngMaxRows = odbcMaxRows
'Устанавливает максимальную ширину поля в этом
' выражении
mtypODBC.IngMaxWidth = odbcMaxWidth
intRet = oSQLSetStmtOption(ByVal_mtypODBC.IngHstmt, _
    SQL_ROWSET_SIZE, mtypODBC.IngRowsetSize)
If intRet <<>> SQL_SUCCESS Then
    Debug.Print "Сообщение: не можем установить размер набора"
    Call rODBCErrorInfo(mtypODBC.IngHenv, _
        mtypODBC.IngHdbc, mtypODBC.IngHstmt)
End If
' Устанавливает тип курсора для этого выражения
intRet = oSQLSetStmtOption(ByVal mtypODBC.IngHstmt, _
    SQL_CURSOR_TYPE, mtypODBC.IngCursor)
If intRet <<>> SQL_SUCCESS Then
    Debug.Print "Сообщение: не можем установить тип курсора"
    Call rODBCErrorInfo(mtypODBC.IngHenv, _
        mtypODBC.IngHdbc, mtypODBC.IngHstmt)
End If
' Устанавливает тип согласования для этого выражения
intRet = oSQLSetStmtOption(ByVal mtypODBC.IngHstmt, _
    SQL_CONCURRENCY, mtypODBC.IngConcur)
If intRet <<>> SQL_SUCCESS Then
    Call rODBCErrorInfo(mtypODBC.IngHenv, _
        mtypODBC.IngHdbc, mtypODBC.IngHstmt)
End If
intRet = oSQLSetStmtOption(ByVal mtypODBC.IngHstmt, _
    SQL_MAX_ROWS, mtypODBC.IngMaxRows)
If intRet <<>> SQL_SUCCESS Then
    Call rODBCErrorInfo(mtypODBC.IngHenv, _
        mtypODBC.IngHdbc, mtypODBC.IngHstmt)
End If
'Присваивает имя курсора выражения
intRet = oSQLSetCursorName(mtypODBC.IngHstmt, _
    "C1", 2)

```

```

If intRet <<>> SQL_SUCCESS Then
    Call rODBCErrorInfo(mtypODBC.IngHenv, _
        mtypODBC.IngHdbc, mtypODBC.IngHstmt)
End If
mtypODBC.strCursorName = rODBCGetCursorName()
' Выполнение SQL выражения
intRet = oSQLExecDirect(ByVal mtypODBC.IngHstmt, _
    strSQL, Len(strSQL))
If intRet <<>> SQL_SUCCESS Then
    Call rODBCErrorInfo(mtypODBC.IngHenv, _
        mtypODBC.IngHdbc, mtypODBC.IngHstmt)
End If
If intRet <<>> SQL_SUCCESS And intRet <<>> SQL_SUCCESS_WITH_INFO Then
    rODBCOpenRecordset = False
    GoTo rODBCOpenRecordsetExit
End If
'Выводит информацию о колонке
intRet = rODBCGetColumnInfo()
If Not intRet Then
    rODBCOpenRecordset = False
    GoTo rODBCOpenRecordsetExit
End If
rODBCOpenRecordsetExit:
Exit Function
rODBCOpenRecordsetErr:
MsgBox Err.Number & ": " & Err.Description, _
    vbCritical, "rODBCOpenRecordset()"
rODBCOpenRecordset = False
Resume rODBCOpenRecordsetExit
End Function

```

Как и большинство API, ODBC API активно использует указатели. Указатели используются для ссылок на объекты, с которыми вы работаете. ODBC приложения работают с тремя типами указателей: окружения, соединения и утверждения. Каждое приложение, которое использует ODBC, начинается с размещения одного указателя окружения (устанавливаемого с помощью SQLAllocEnv) и заканчивается освобождением этого указателя (SQLFreeEnv). Указатель окружения - это родительский или главный указатель, с которым непосредственно связаны другие ресурсы ODBC, размещаемые для приложения.

Для того чтобы исключить появление незавершенных результатов и освободить указатель SQL выражения, используйте следующее выражение, предварительно объявив функцию с помощью команды **Declare**:

```

Declare Function oSQLFreeStmt Lib "odbc32.dll" _
Alias "SQLFreeStmt" (ByVal hstmt As Long, ByVal fOption As Integer) As Integer
rc = oSQLFreeStmt (mtypODBC.IngHstmt, SQL_CLOSE)

```

Следующие функции служат для освобождения указателя соединения и окружения:

```

Declare Function oSQLFreeConnect Lib "odbc32.dll" _
Alias "SQLFreeConnect" (ByVal hdbc As Long) As Integer
Declare Function oSQLFreeEnv Lib "odbc32.dll" _
Alias "SQLFreeEnv" (ByVal henv As Long) As Integer

```

Как правило, любой сеанс работы с использованием ODBC API должен заканчиваться вызовом этих функций. Вызывая их из Access или Visual Basic, вы можете оформить это примерно таким образом, как показано в следующем примере ниже описании функции:

```

Function rODBCDisconnect()
' Отсоединение от источника данных (Data Source)
Dim intRet As Integer
On Error GoTo rODBCDisconnectErr
intRet = oSQLDisconnect(ByVal mtypODBC.IngHdbc)
If intRet <<>> SQL_SUCCESS Then
    Call rODBCErrorInfo(mtypODBC.IngHenv, _

```

```

        mtypODBC.IngHdbc, SQL_NULL_HSTMT)
    End If
    intRet = oSQLFreeConnect(ByVal mtypODBC.IngHdbc)
    If intRet <<>> SQL_SUCCESS Then
        Call rODBCErrorInfo(mtypODBC.IngHenv, _
            mtypODBC.IngHdbc, SQL_NULL_HSTMT)
    End If
    intRet = oSQLFreeEnv(ByVal mtypODBC.IngHenv)
    If intRet <<>> SQL_SUCCESS Then
        Call rODBCErrorInfo(mtypODBC.IngHenv, _
            SQL_NULL_HDBC, SQL_NULL_HSTMT)
    End If
    rODBCDisconnectExit:
    Exit Function
rODBCDisconnectErr:
    MsgBox Err.Number & ": " & Err.Description, _ vbCritical, "rODBCDisconnect()"
    rODBCDisconnect = False
    Resume rODBCDisconnectExit
End Function

```

Как видно из вышеизложенного, функции ODBC API можно разделить на несколько типов, в зависимости от задач, для которых они предназначены.

В табл. 8.6 перечислены базисные функции или функции ядра ODBC с пояснениями их назначения (см. также рис. 8.12). При этом функции объединены по признаку их назначения.

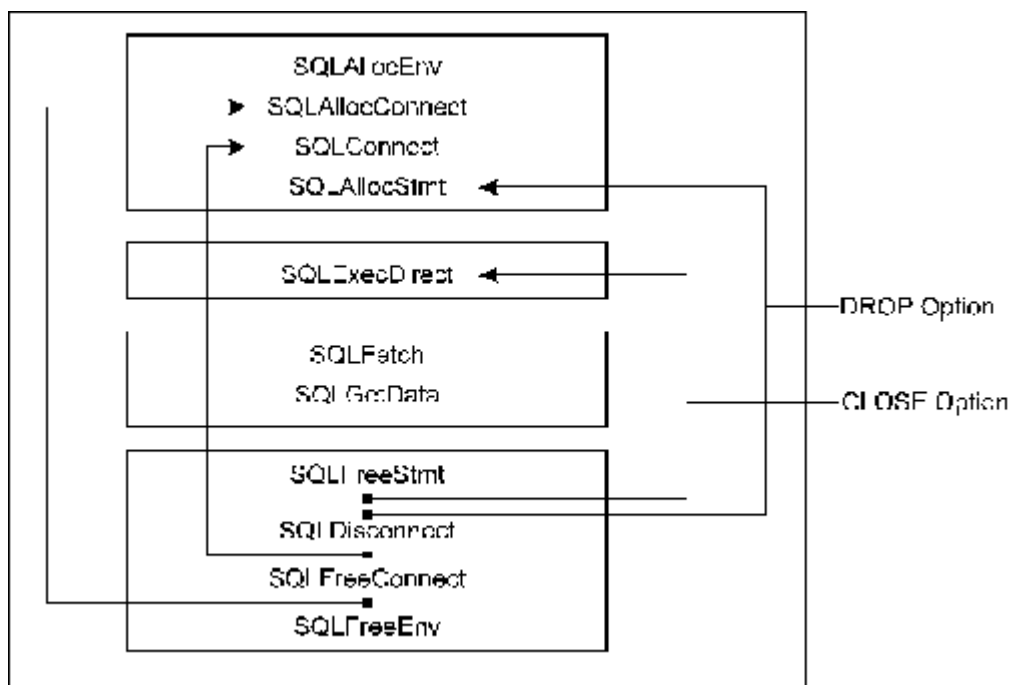


Рис. 8.12.

Таблица 8.6. Базисные функции ODBC API

Назначение	Функция	Описание
Соединение с источником данных	SQLAllocEnv	Получает указатель окружения. Одно окружение может служить для создания нескольких соединений.
^	SQLAllocConnect	Получает указатель соединения.
^	SQLConnect	Соединяется с указанным драйвером, используя имя источника данных, идентификатор пользователя и пароль.
Подготовка SQL запросов	SQLAllocStmt	Размещает указатель выражения.

	^	SQLPrepare	Подготавливает SQL выражение для дальнейшего использования.
	^	SQLGet-CursorName	Возвращает имя, связанное с указателем выражения.
	^	SQLSet-CursorName	Устанавливает имя курсора.
Выполнение запросов		SQLExecute	Выполняет заранее подготовленный запрос.
	^	SQLExec-Direct	Выполняет запрос.
Выборка результатов и информации о результатах		SQLRow-Count	Возвращает количество записей, задействованных в операциях вставки, удаления, модификации.
	^	SQLNum-ResultCol	Возвращает количество колонок в выбранном наборе данных.
	^	SQLDescribe-Col	Описывает колонку в выбранном наборе данных.
	^	SQLCol-Attributes	Описывает атрибуты колонки в выбранном наборе данных.
	^	SQLBindCol	Присваивает место в памяти для колонки в выбранном наборе данных и указывает ее тип данных.
	^	SQLFetch	Возвращает несколько наборов данных.
Окончание работы выражения		SQLFreeStmt	Заканчивает процесс работы выражения.
	^	SQLCancel	Прерывает работу выражения.
	^	SQLTransact	Завершает или откатывает транзакцию.
Окончание работы соединения		SQL-Disconnect	Закрывает транзакцию.
	^	SQLFreeEnv	Удаляет указатель окружения.
	^	SQLFree-Connect	Удаляет указатель соединения.

Ниже приводится пример объявления функций ODBC API, которые вы можете применять в своих программах, если, конечно, в качестве среды разработки используется система, поддерживающая вызов DLL функций:

```

Declare Function oSQLRowCount Lib "odbc32.dll" _
Alias "SQLRowCount" (ByVal hstmt As Long, pcrow As Long) As Integer
Declare Function oSQLBindCol Lib "odbc32.dll" _
Alias "SQLBindCol" (ByVal hstmt As Long, ByVal icol As Integer, _
ByVal fCType As Integer, rgbValue As Any, _
ByVal cbValueMax As Long, pcbValue As Long) As Integer
Declare Function oSQLColAttributes Lib "odbc32.dll" _
Alias "SQLColAttributes" (ByVal hstmt As Long, ByVal icol As Integer, _
ByVal fCType As Integer, rgbDesc As Any, ByVal cbDescMax As Integer, _
pcbDesc As Integer, pfDesc As Long) As Integer
Declare Function oSQLDescribeCol Lib "odbc32.dll" _
Alias "SQLDescribeCol" (ByVal hstmt As Long, _
ByVal icol As Integer, ByVal szColName As String, _
ByVal cbColNameMax As Integer, pcbColName As Integer, _
pfSQLType As Integer, pcbColDef As Long, _
pibScale As Integer, pfNullable As Integer) As Integer
Declare Function oSQLDisconnect Lib "odbc32.dll" _
Alias "SQLDisconnect" (ByVal hdbc As Long) As Integer
Declare Function oSQLDriverConnect Lib "odbc32.dll" _
Alias "SQLDriverConnect" (ByVal hdbc As Long, _

```

```

ByVal hwnd As Long, ByVal szConnStrIn As String, _
ByVal cbConnStrIn As Integer, _
ByVal szConnStrOut As String, ByVal cbConnStrOutMax As Integer, _
pcbConnStrOut As Integer, ByVal fDriverCompletion As Integer) As Integer
Declare Function oSQLError Lib "odbc32.dll" _
Alias "SQLError" (ByVal henv As Long, ByVal hdbc As Long, _
ByVal hstmt As Long, ByVal szSqlState As String, _
pfNativeError As Long, ByVal szErrorMessage As String, _
ByVal cbErrorMsgMax As Integer, pcErrorMsg As Integer) As Integer

```

Объявленная выше последняя функция служит для обработки ошибок, которые могут возникнуть и возникают при использовании ODBC API. ODBC поддерживает стандартную модель обработки ошибок. Каждая функция ODBC возвращает некий код, одним из которых может быть **SQL_ERROR**. Чтобы получить больше информации об ошибке, приложение вызывает функцию **SQLError()**.

Драйвер хранит информацию об ошибке в структурах **henv**, **hdbc** и **hstmt** и возвращает эту информацию приложению, когда приложение вызывает **SQLError()**. Каждая функция может вызвать ноль или больше ошибок.

Приложение обычно вызывает функцию **SQLError()**, когда предыдущий вызов ODBC функции возвращает **SQL_ERROR** или **SQL_SUCCESS_WITH_INFO**. Приложение может тем не менее вызвать **SQLError()** после вызова любой ODBC функции.

Функция **SQLError()** возвращает следующую информацию:

- **SQLSTATE** - стандартный идентификатор ошибки.
- **Native Error Code** - код ошибки, свойственный данному источнику данных.
- **Error Message Text** - описание ошибки.

Ошибки сохраняются для текущего указателя до тех пор, пока данный указатель не будет использован в вызове следующей функции. К примеру, ошибки на **hstmt** для текущей функции очищаются, как только другая функция будет выполнена с использованием такого же указателя. Ошибки, хранимые для данного указателя, никогда не очищаются в результате вызова функции с использованием указателя другого, хотя и родственного типа. Например, ошибки на **hdbc** не очищаются, когда вызов делается к родственной **hstmt**.

Функция **SQLError()** возвращает ошибку из структуры, связанной с самым правым ненулевым аргументом указателя. Приложение запрашивает информацию об ошибке в следующем порядке:

- Чтобы получить ошибки, связанные с окружением, приложение передает соответствующий **henv** и включает **SQL_NULL_HDBC** и **SQL_NULL_HSTMT** в **hdbc** и **hstmt** соответственно. Драйвер возвращает статус ошибки ODBC функции, вызываемой самой последней с тем же самым **henv**.
- Для вывода ошибок, связанных с соединением, приложение передает соответствующий **hdbc** плюс **hstmt**, равный **SQL_NULL_HSTMT**. В таком случае драйвер игнорирует аргумент **henv**. Драйвер возвращает статус ошибки функции ODBC, вызванной самой последней с **hdbc**.
- Для отслеживания ошибок, связанных с выражением, приложение передает соответствующий указатель **hstmt**. Если вызов **SQLError()** содержит правильный указатель **hstmt**, драйвер игнорирует аргументы **hdbc** и **henv**. Драйвер возвращает статус ошибки самой последней функции ODBC, вызванной с указателем **hstmt**.
- Для отслеживания нескольких ошибок, вызываемых функцией, приложение обращается к **SQLError()** несколько раз. Для каждой ошибки драйвер возвращает **SQL_SUCCESS** и удаляет эту ошибку из списка доступных ошибок.

Когда отсутствует дополнительная информация по самому правому ненулевому указателю, функция **SQLError()** возвращает **SQL_NO_DATA_FOUND**.

```

Declare Function oSQLExtendedFetch Lib "odbc32.dll" _
Alias "SQLExtendedFetch" (ByVal hstmt As Long, _
ByVal fFetchType As Long, ByVal irow As Integer, _
pcrow As Long, rgfRowStatus As Integer) As Integer
Declare Function oSQLFetch Lib "odbc32.dll" _
Alias "SQLFetch" (ByVal hstmt As Long) As Integer
Declare Function oSQLGetCursorName Lib "odbc32.dll" _
Alias "SQLGetCursorName" (ByVal hstmt As Long, _
ByVal szCursor As String, ByVal cbCursorMax As Integer, _

```

```

pcbCursor As Integer) As Integer
Declare Function oSQLGetData Lib "odbc32.dll" _
Alias "SQLGetData" (ByVal hstmt As Long, ByVal icol As_ Integer, _
ByVal fCType As Integer, rgbValue As Any, _
ByVal cbValueMax As Long, pcbValue As Long) As Integer
Declare Function oSQLGetInfo Lib "odbc32.dll" _
Alias "SQLGetInfo" (ByVal hdbc As Long, _
ByVal flInfoType As Integer, ByRef rgbInfoValue As Any, _
ByVal cbInfoMax As Integer, cbInfoOut As Integer) As_ Integer
Declare Function oSQLGetStmtOption Lib "odbc32.dll" _
Alias "SQLGetStmtOption" (ByVal hstmt As Long, _
ByVal fOption As Integer, pvparam As Any) As Integer
Declare Function oSQLNumResultCols Lib "odbc32.dll" _
Alias "SQLNumResultCols" (ByVal hstmt As Long, pccol As_ Integer) As Integer
Declare Function oSQLSetCursorName Lib "odbc32.dll" _
Alias "SQLSetCursorName" (ByVal hstmt As Long, _
ByVal szCursor As String, ByVal cbCursor As Integer) As_ Integer
Declare Function oSQLSetPos Lib "odbc32.dll" _
Alias "SQLSetPos" (ByVal hstmt As Long, _
ByVal irow As Integer, ByVal fOption As Integer, _
ByVal fLock As Integer) As Integer
Declare Function oSQLSetStmtOption Lib "odbc32.dll" _
Alias "SQLSetStmtOption" (ByVal hstmt As Long, _
ByVal fOption As Integer, ByVal bparam As Long) As_ Integer

```

8.5. Remote Data Objects

Технология Remote Data Objects (RDO) - объекты для доступа к внешним данным - доступна, если на компьютере установлена версия Visual Basic 4.0 Enterprise. Очень мало, а в некоторых случаях и совсем не уступающая по скорости доступа к данным, по отношению к прямому использованию ODBC API, данная технология значительно проще в использовании. Особенно тем, кто знаком с DAO - набором объектов для доступа к данным.

В этом параграфе мы рассмотрим практические вопросы использования технологии RDO в приложении клиент-сервер.

RDO представляет собой тонкую прослойку кода над ODBC API и менеджером драйверов, которая устанавливает соединения, создает наборы данных и курсоры, выполняет другие сложные процедуры, требуя минимума ресурсов рабочей станции, так как все процессы происходят на сервере.

Есть определенные различия при визуальной работе с RDO в Visual Basic и Access. В Visual Basic у вас есть возможность воспользоваться элементом управления ActiveX - RemoteControl, который еще в большей степени скрывает сложность работы с внешними данными. Вам достаточно установить лишь несколько свойств этого объекта, и вы получаете доступ к необходимой информации.

Используя RDO или объект RemoteControl, вы обходите процессор данных. При этом вы можете иметь доступ к внешним данным любого формата, хотя лучше использовать метод для доступа к данным на сервере таких баз данных, как MS SQL Server или Oracle. При работе с RDO вы можете использовать как синхронный, так и асинхронные процессы, поэтому при большой выборке данных с сервера ваше приложение не будет заблокировано на период получения данных.

В табл. 8.7 приводятся эквиваленты объектов DAO для объектов RDO.

RDO оперирует понятиями "строка", а не "запись" и "колонка", а не "поле". Так же как DAO, все объекты в RDO содержатся в коллекциях, исключение составляет только rdoEngine. Иерархия объектов RDO приведена на рис. 8.13. Когда RDO инициализируется в первый раз, то создается экземпляр объекта rdoEngine и коллекция rdoEnvironments, состоящая из одного объекта rdoEnvironments(0).

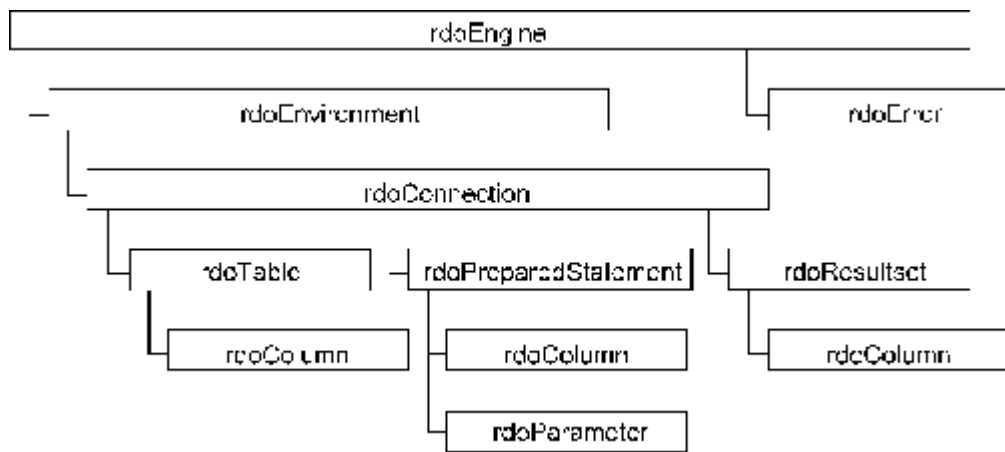


Рис. 8.13.

Для установки связи с источником данных ODBC и необходимой базой данных необходимо создать объект `rdoConnection`. Если у вас нет необходимого источника данных, то вы можете создать его с помощью метода `rdoRegisterDataSource` или с помощью диалоговых средств Администратора ODBC.

Таблица 8.7. Соответствие объектов RDO объектам DAO

Объект RDO	Эквивалентный объект DAO
<code>rdoEngine</code>	<code>DBEngine</code>
Отсутствует	User, Group
<code>rdoEnvironment</code>	Workspace
<code>rdoConnection</code>	Database
<code>rdoTable</code>	TableDef
Отсутствует	Index
<code>rdoResultset</code>	Recordset
Forward-only - type	Похоже на Forward-Only Snapshot
<code>rdoColumn</code>	Field
<code>rdoPreparedStatement</code>	QueryDef

Существует два подхода к выполнению запросов и созданию результирующего набора данных:

- Использование запроса, который будет применяться только один раз. В этом случае воспользуйтесь методами `OpenResultSet` или `Execute`, выполняющими SQL запрос для существующего объекта `rdoConnection` и создающими объект `rdoResultset` или выполняющими запрос действия.
- Использование запроса, который будет применяться неоднократно и дополнительно может иметь параметры. Воспользуйтесь методом `CreatePreparedStatement` для создания объекта `rdoPreparedStatement`, который может использоваться в любое время, когда он снова понадобится и для которого можно изменять значения параметров перед каждым повторным использованием. После того как объект `rdoPreparedStatement` будет создан, используйте методы `OpenResultSet` и `Execute` с объектом `rdoPreparedStatement`, для того чтобы создать объект `rdoResultset` или выполнить запрос действия. Для того чтобы изменить параметры, используйте установки объекта `rdoParameter`.

Используя аргументы метода `OpenResultSet` или свойства объекта `rdoPreparedStatement`, вы можете установить тип курсора и другие атрибуты объекта `rdoResultset`.

Перед тем как начать работать с RDO, необходимо, используя команду *References* меню *Tools*, подключить объекты RDO к вашему проекту. В списке доступных объектов они так и будут называться - Microsoft Remote Data Object. Если вы работаете в Access, то достаточно наличия этих объектов на компьютере, но, как указывалось выше, для этого необходимо установить Visual Basic Enterprise Edition. После того как эти условия выполнены, вы можете свободно обращаться ко всем методам и свойствам этих объектов.

Самым верхним в иерархии объектов является `rdoEngine`. Вспомните об объекте `dbEngine` из DAO. С помощью следующей строчки вы уже обращаетесь к RDO:

`rdoEngine.rdoDefaultCursorDriver = rdUseOdbc`

Объект `rdoEngine` имеет следующие свойства:

- `rdoDefaultCursorDriver` - может принимать три значения, которые являются предопределенными константами:

Константа	Значение	Описание
<code>rdUseIfNeeded</code>	0	ODBC драйвер будет использовать соответствующий тип курсора. Курсоры сервера будут использоваться, если они доступны.
<code>rdUseODBC</code>	1	В данном случае RDO использует библиотеку курсоров ODBC. Наиболее подходящий путь при работе с выборками небольшого объема.
<code>RdUseServer</code>	2	Драйвер ODBC использует курсоры сервера. Подходит для большинства операций при обработке больших массивов данных, но приводит к резкому возрастанию сетевого трафика.

Курсором является логический набор данных, управляемый источником данных или Диспетчером ODBC.

- `rdoDefaultErrorThreshold` - данное свойство устанавливает или возвращает установку по умолчанию для свойства `ErrorThreshold` объекта `rdoPreparedStatement`. Суть использования данного свойства заключается в том, что каждая ошибка имеет свойство `Number`. Если свойство `Number` больше значения свойства `ErrorThreshold`, то ошибка не генерируется, в противном случае генерируется перехватываемая ошибка, которая либо заканчивает работу приложения, либо обрабатывается, если это предусмотрено в коде или системе. Если установить это свойство равным -1, то, соответственно, никакого объекта для отсечения перехватываемых ошибок не будет.

- `rdoDefaultLoginTimeout` - возвращает или устанавливает количество секунд, в течение которых ODBC драйвер прекратит попытку установить соединение с источником данных и возвратит перехватываемую ошибку. Значение по умолчанию - 15 с. Если это значение равно 0, то попытка установить соединение с источником данных, используя текущую процедуру регистрации, будут продолжаться неопределенное время.

Свойства `rdoDefaultPassword` и `rdoDefaultUser` устанавливают пароль и пользователя по умолчанию для всякого вновь создающегося объекта `rdoEnvironment`.

Объект `rdoEngine` имеет два метода. Метод `rdoRegisterDataSource` служит для занесения в Регистр Windows информации о вновь создаваемом источнике данных ODBC. Таким образом вы можете создавать источники данных, минуя диалоговое окно Администратора ODBC. Это очень удобно при распространении вашего приложения. Хотя есть и другие способы внести информацию о новом источнике данных в Регистр Windows.

Приведем синтаксис этого метода:

`rdoRegisterDataSource cDataSourceName, cDriverName, ISilent, cAttributes`

где

- *cDataSourceName* - имя, которое вы хотите присвоить источнику данных. В дальнейшем остальные методы, которым необходимо связаться с базой данных, для которой вы создаете источник данных, будут использовать это имя. Например, метод `OpenConnection`.
- *cDriverName* - строковое выражение, являющееся именем драйвера, зарегистрированным в Регистре Windows. При этом обращаем внимание, что драйвер уже должен быть установлен.
- *ISilent* - логическое выражение, которое указывает, будет ли появляться диалог ODBC, в котором вы укажете специфичную для данного драйвера информацию. Если вы укажете это значение равным `True`, то вам надо будет указать всю необходимую информацию в аргументе *cAttributes*.
- *cAttributes* - строковое выражение, в котором вы указываете дополнительную информацию для драйвера. Для каждого драйвера эта информация специфична. Параметры, которые необходимо описать, вы можете посмотреть в Регистре Windows, найдя описание источника данных, которые устанавливает связь с базой данных подобного формата. Например так, как это показано на рис. 8.14.

Для работы с внешними данными RDO использует объект окружения (*rdoEnvironment*). При этом одно окружение в коллекции создается автоматически. Этого окружения вполне достаточно для работы с данными, так как с помощью него вы можете создать сколько угодно соединений и одновременно редактировать таблицы из нескольких баз данных, даже разного формата. Но иногда необходимо обрабатывать более сложные ситуации, в которых надо вести несколько транзакций одновременно. Тогда имеет смысл создавать дополнительные объекты *rdoEnvironment*. Здесь необходимо отметить, что ODBC не поддерживает возможности вложенных транзакций. Выход из положения можно найти, если приложение, с данными которого вы работаете, поддерживает вложенные транзакции. Тогда вы можете использовать SQL выражения, которые будут передаваться на сервер и создавать сложные вложенные и пересекающиеся транзакции.

Объект *rdoEnvironment* создается с помощью метода *rdoCreateEnvironment* и имеет следующий синтаксис:

```
set Variable = rdoCreateEnvironment(Name, User, Password)
```

где

- *Variable* - объектная переменная, которая ссылается на объект *rdoEnvironment*.
- *Name* - строковая переменная, которая становится уникальным именем объекта *Environment*.
- *User* - имя пользователя объекта *rdoEnvironment*.
- *Password* - пароль пользователя.

Приведенная ниже программа демонстрирует установку и чтение некоторых из вышеприведенных свойств и использование методов *rdoRegisterDataSource* и *rdoCreateEnvironment*.

```
Const rServerDSN = "ToAutostore"
Const rServerUser = "UID=sa;DATABASE=autostore"
Dim grdfEnv As rdo.rdoEnvironment
Dim grdfConn As rdo.rdoConnection
Dim sqlAttr As String
strAttr = "Description=Connect to autostore" & _
Chr$(13) & "OemToAnsi=No" & _
Chr$(13) & "Address=\\MAINSERVER\AUTOSTORE\" & Chr$(13) & _
"Database=Autostore"
rdoEngine.rdoRegisterDatasource rServerDSN, _ "SQL Server", True, sqlAttr
If grdfEnv Is Nothing Then
    rdoEngine.rdoDefaultCursorDriver = rdUseOdbc
    Set grdfEnv = rdoEngine.rdoCreateEnvironment _
    ("", "", "")
    Set grdfConn = grdfEnv.OpenConnection _
    (rServerDSN, rdDriverNoPrompt, False, _
    rServerUser)
    grdfConn.QueryTimeout = 0
End If
```

Коллекция *rdoEnvironments* содержит в себе все объекты *rdoEnvironment* объекта *rdoEngine*. Она имеет всего один метод *Item*, который устанавливает доступ к конкретному объекту в зависимости от его порядкового номера в коллекции. То есть вы можете обращаться к первому объекту в коллекции с помощью метода *Item*, например:

```
Set grdfConn = rdoEnvironments.Item(0).grdfEnv.OpenConnection _
(rServerDSN, rdDriverNoPrompt, False, _
rServerUser)
```

В то же время следующий пример эквивалентен предыдущему:

```
Set grdfConn = rdoEnvironments(0).grdfEnv.OpenConnection _
(rServerDSN, rdDriverNoPrompt, False, _
rServerUser)
```

Коллекция *rdoEnvironments* имеет свойство *Count*, которое используется только для чтения и возвращает количество объектов в коллекции.

Объект *rdoEnvironment* содержит три метода для работы с транзакциями. Это методы *BeginTrans*, *CommitTrans* и *RollBackTrans*. Эти методы очень похожи на соответствующие методы

объекта *Workspace* из *DAO*. Но имеет смысл напомнить еще раз, что нельзя использовать вложенные транзакции. В то же время вы можете управлять транзакциями на сервере с помощью SQL команд **BEGIN TRANSACTION**, **COMMIT TRANSACTION**, **ROLLBACK TRANSACTION**, передавая их на сервер с помощью запросов действия.

BEGINTRANS начинает транзакцию.

COMMITTRANS прекращает текущую транзакцию и сохраняет все изменения.

ROLLBACKTRANS заканчивает текущую транзакцию и восстанавливает базы данных в текущем объекте *rdoEnvironment* в то состояние, в котором они находились до начала транзакции.

Объект *rdoEnvironment* имеет метод *Close*. При работе с этим методом необходимо помнить, что вы не можете закрыть окружение по умолчанию, то есть объект *rdoEnvironments(0)*. Если текущий объект *rdoEnvironment* содержит открытые соединения, которые используются объектами *rdoResulset*, то после закрытия объекта *rdoEnvironment* все текущие транзакции прекращаются и базы данных возвращаются в первоначальное состояние.

Метод *OpenConnection* служит для создания и открытия соединений, которые являются единственной коллекцией, содержащейся в объекте *rdoEnvironment* и объекты которой используются для доступа к данным в базе данных, указанной в источнике данных. Этот метод имеет следующий синтаксис:

```
Set Connection = Environment.OpenConnection(DataSourceName[,  
Prompt[, Readonly[, Connect]]])
```

Connection - объектная переменная, которая ссылается на объект *rdoConnection*.

Environment - выражение, которое обозначает существующее окружение.

DataSourceName - строковое выражение, обозначающее имя источника данных, так как оно зарегистрировано в Реестре *Windows*. При этом если не указать его, то есть ввести пустую строку или указать несуществующий источник данных, то будет выведено диалоговое окно *ODBC* для выбора источника данных из списка существующих. Тем не менее, если аргумент *Prompt* имеет значение *rdDriverNoPrompt*, будет сгенерирована перехватываемая ошибка и никакого диалогового окна выведено не будет. В то же время можно ввести имя источника данных в аргументе *Connect* в параметре *DataSourceName*.

Prompt может принимать одно из четырех значений предопределенных констант. В зависимости от этого значения диспетчер *ODBC* драйверов требует от пользователя ввода *DataSourceName* (имени источника данных), имени пользователя и пароля.

Значение аргумента	Описание
<i>rdDriverNoPrompt</i>	Диспетчер драйверов использует информацию из аргументов <i>DataSourceName</i> и <i>Connect</i> для построения строки соединения. Если необходимая информация не будет обеспечена, то метод <i>OpenConnection</i> возвратит ошибку.
<i>RdDriverPrompt</i>	Диспетчер драйверов выводит диалог <i>ODBC</i> и строит строку соединения по информации, введенной пользователем в этом диалоге.
<i>rdDriverComplete</i>	Диспетчер драйверов выводит диалог <i>ODBC</i> только в случае недостатка информации в аргументах <i>DataSourceName</i> и <i>Connect</i> .
<i>rdDriverCompleteRequired</i>	Диспетчер драйверов ведет себя так же, как в случае, когда данный аргумент равен <i>rdDriverComplete</i> , за исключением того, что в диалоге <i>ODBC</i> отключены элементы,

предоставляющие
информацию, которая
больше не нужна.

- **Readonly** - определяет, будет ли соединение открыто для доступа только для чтения или для чтения и записи.
- **Connect** - строковое выражение, используемое для открытия баз данных. Эта строка составляет ODBC аргументы соединения и зависит от конкретного используемого драйвера.

Иногда могут возникнуть причины, по которым вы не сможете установить соединение, среди них такие, как отсутствие прав доступа к источнику данных, неправильное соединение сети, отсутствие или отключение источника данных. Некоторые серверы баз данных имеют ограничение на количество соединений по разным причинам, в том числе и ограничение по ресурсам.

Создав объект `rdoConnection`, вы можете использовать его для

- Создания объектов `rdoResultset` или `rdoPreparedStatement`, используя методы `OpenResultset` или `CreatePreparedStatement`, в зависимости от решаемых задач.
- Доступа к таблицам базы данных и колонкам каждой таблицы, используя объекты `rdoTable` и `rdoColumn` в коллекциях `rdoConnections(0).rdoTables` и `rdoTables(n).rdoColumns`.
- Начала, завершения или отката транзакции, используя методы `Begin-Trans`, `CommitTrans` и `RollbackTrans`.
- Отсоединения от источника данных и освобождения ресурсов, используя метод `Close`.

Объект `rdoEnvironment` имеет несколько свойств, которые перечислены в следующей таблице.

Свойство	Описание
<code>CursorDriver</code>	Служит для установки и чтения значения, устанавливающего тип курсора, который будет создан. Может иметь три значения: <code>rdUsedIfNeeded</code> , <code>rdUseODBC</code> , <code>rdUseServer</code> .
<code>hEnv</code>	Возвращает значение указателя ODBC окружения, которое можно использовать для вызова ODBC API функций.
<code>LoginTimeout</code>	При использовании баз данных ODBC возможны задержки из-за сетевого трафика или напряженного использования источника данных ODBC. Чтобы избежать неопределенно долгого ожидания, можно установить время, после которого диспетчер драйверов ODBC сгенерирует ошибку.
<code>Name</code>	Имя объекта.
<code>Password</code>	Возвращает пароль, используемый во время создания объекта <code>rdoEnvironment</code> .
<code>UserName</code>	Возвращает или устанавливает имя пользователя объекта <code>rdoEnvironment</code> .

Следующая коллекция объектов, которая входит в объект `rdoEngine`, называется `rdoErrors` и служит для обработки ошибок при работе с RDO.

Всякий раз, когда диспетчер ODBC пытается выполнить запрос RDO, может возникнуть ошибка. Подобные ошибки могут иметь различные причины и вызывать различные последствия для вашей программы, вплоть до полного прекращения процесса выполнения запроса или даже вашей процедуры или функции. Как только ошибки возникли, информация о них помещается в коллекцию `rdoErrors`. Далее появляется возможность исследовать каждый объект `rdoError` коллекции `rdoErrors` на предмет причины возникновения ошибки и выполнения дальнейших

действий. Visual Basic также создает перехватываемую ошибку во время возникновения ошибочных ситуаций. Поэтому имеет смысл использовать конструкцию

On Error Goto <<метка>>

для обработки информации, содержащейся в объектах коллекции `rdoError` и для принятия последующих решений. Два свойства - `rdoDefaultErrorTreshold` и `ErrorTreshold` - позволяют понизить или повысить ограничения на ситуации, которые могут вызвать фатальные ошибки.

Добавим в предыдущий пример строчки, которые будут заниматься обработкой ошибок:

```
Function rdfInitialize() As Boolean
Const rServerDSN = "ToAutostore"
Const rServerUser = "UID=sa;DATABASE=autostore"
Dim errX As rdo.rdoError
Dim grdfEnv As rdo.rdoEnvironment
Dim grdfConn As rdo.rdoConnection
Dim sqlAttr As String
strAttr = "Description=Connect to autostore" & _
Chr$(13) & "OemToAnsi=No" & _
Chr$(13) & "Address=\\MAINSERVER\AUTOSTORE\" & Chr$(13) & _
"Database=Autostore"
rdoEngine.rdoRegisterDatasource rServerDSN, _ "SQL Server", _
True, sqlAttr
If grdfEnv Is Nothing Then
rdoEngine.rdoDefaultCursorDriver = rdUseOdbc
Set grdfEnv = rdoEngine.rdoCreateEnvironment _
("","","")
Set grdfConn = grdfEnv.OpenConnection _
(rServerDSN, rdDriverNoPrompt, False, _
rServerUser)
grdfConn.QueryTimeout = 0
End If
rdfInitializeExit:
DoCmd.Hourglass False
Exit Function
rdfInitializeErr:
For Each errX In rdoEngine.Errors
MsgBox "Ошибка " & errX.Number & " вызвана " _
& errX.Source & ": " & errX.Description, _
vbCritical, "rdfInitialize()"
Next errX
rdfInitialize = False
Resume rdfInitializeExit
End Function
```

Коллекция `rdoErrors` имеет два метода. Метод `Clear` служит для удаления всех объектов `rdoError` из коллекции. Метод `Item` предназначен для доступа к объектам `rdoError` по их индексу. Впрочем, следующие строчки эквивалентны:

```
myRdoError=rdoEngine.rdoErrors.Item(1)
myRdoError=rdoEngine.rdoErrors(1)
```

Свойство `Count` возвращает количество объектов `rdoError` в коллекции. Следующая таблица описывает свойства объекта `rdoError`.

Свойство	Описание
<code>Description</code>	Возвращает строковое выражение, содержащее описание ошибки.
<code>HelpContext</code>	Если в свойстве <code>HelpFile</code> указан файл помощи Microsoft Windows, свойство <code>HelpContext</code> используется для того, чтобы автоматически выводить раздел помощи, который он идентифицирует.

Help-File	Указывает полный путь к файлу помощи.
Source	Возвращает строковое выражение. При возникновении ошибки во время операций ODBC объект <code>rdoError</code> добавляется в коллекцию <code>rdoErrors</code> . Если ошибка возникла внутри RDO, возвращаемое значение начинается с выражения "MSRDO32". Объектный класс, который явился инициатором ошибки, также может быть добавлен к значению свойства <code>Source</code> .
SQLRet-Code	Возвращает код ошибки последней операции RDO. Значение имеет тип <code>Long</code> и может равняться одной из следующих констант: <code>rdSQLSuccess</code> - операция завершилась успешно; <code>rdSQLSuccessWithInfo</code> - операция завершилась успешно и доступна дополнительная информация; <code>rdSQLNoDataFound</code> - нет никаких доступных дополнительных данных; <code>rdSQLException</code> - ошибка, случившаяся во время выполнения операции; <code>rdSQLInvalidHandle</code> - предоставленный указатель указан неверно.
SQL-State	Символьная строка, возвращаемая свойством <code>SQLState</code> , состоит из двухсимвольного значения класса, за которым следует трехсимвольное значение подкласса. Значение класса "01" указывает на предупреждение и сопровождается кодом, возвращаемым <code>rdSQLSuccessWithInfo</code> .

Следующим объектом в иерархии RDO является `rdoConnection`, который представляет собой физическое соединение с сервером. Объект `rdoConnection` имеет методы для работы с транзакциями, но в данном случае налицо явный пример полиморфизма, так как здесь диапазон действия этих методов ограничивается ODBC процессами, использующими указатель текущего соединения. То есть, если вы применяете параллельные транзакции, каждая из которых использует свое соединение или объект `rdoConnection`, то можете завершить каждую транзакцию отдельно, применяя методы `CommitTrans` каждого объекта по очереди. Если необходимо завершить все транзакции в текущем окружении, то используйте метод `CommitTrans` (или `RollbackTrans`) объекта `rdoEnvironment`.

С помощью объекта `rdoConnection` мы имеем доступ к трем объектам, находящимся на последних уровнях иерархии RDO. Объект `rdoTable` предоставляет таблицы и представления (View), хранящиеся в базе данных. Объект `rdoTable` позволяет нам узнать много полезной информации о таблице или представлении на сервере. С его помощью можно получить сведения о типе таблицы, количестве записей и возможности модификации ее данных, что является достаточно важным свойством, так как представления на сервере очень часто не позволяют изменять содержимое строк и колонок, которые они порождают. В то же время для объекта `rdoTable` нет методов для перемещения по записям, поэтому достаточно затруднительно вывести информацию о записи, которая не является первой. Объекты `rdoTable` входят в коллекцию `rdoTables`. При этом мы можем обращаться к любому ее объекту по его имени, то есть следующим образом:

```
Set myrdEnv =rdoEngine.rdoEnvironment(0)
Set myrdConn=myrdEnv.OpenConnection("Toautostore", _ rdDriverNoPrompt, False)
Set myrdTable = myrdConn.rdoTables("account")
```

Последнюю строчку можно переписать следующим образом:

```
Set myrdTable = myrdConn.rdoTables!account
```

Если вы знаете порядковый номер объекта `rdoTable` в коллекции `rdoTables`, то возможно и такое обращение:

```

Set myrdEnv =rdoEngine.rdoEnvironment(0)
Set myrdConn=myrdEnv.OpenConnection("Toautostore", _ rdDriverNoPrompt, False)
Set myrdTable = myrdConn.rdoTables(2)

```

Вторым объектом на этом уровне иерархии, который мы рассмотрим, будет объект `rdoResults`. При разработке интерфейса конечного пользователя именно этот объект является самым главным. Следующий код показывает простейший пример получения и использования этого объекта.

Создайте в Visual Basic 4.0 Enterprise Edition форму, в которой разместите текстовое поле с названием `text1`.

```

Private Sub Form_Load()
    'Создается переменная объекта окружения -
    'соответствует Workspace
    Dim RDE As rdo.rdoEnvironment
    ' Создается переменная соединения - объекта,
    ' служащего для скрытия
    ' сложной работы функций ODBC API,
    ' и который позволяет нам подсоединяться
    ' к данным любого формата при наличии,
    ' естественно, соответствующего
    ' драйвера
    Dim rdConn As rdo.rdoConnection
    ' Ниже создаются объекты таблицы,
    ' колонки, набора данных, ошибки
    Dim rdoTab As rdo.rdoTable
    Dim rdoCol As rdo.rdoColumn
    Dim rdoRst As rdo.rdoResultset
    Dim errX As rdoError
    ' Свойство rdoDefaultCursorDriver определяет, какой
    ' курсор будет
    ' использоваться для перемещения по данным -
    ' курсор ODBC или сервера
    ' в нашем случае мы выбрали курсор
    ' сервера, так как при работе с данными
    ' большого набора это лучший выбор
    rdoEngine.rdoDefaultCursorDriver = rdUseServer
    'Инициализация переменной окружения
    Set RDE = rdoEngine.rdoCreateEnvironment("", "", "")
    'Инициализация переменной соединения
    Set rdConn = RDE.OpenConnection("myteach",_rdDriverNoPrompt, False)
    'Инициализация обработчика ошибок
    On Error GoTo rdflInitializeErr
    ' Данная строка помещена из методических
    ' соображений,
    ' потому что вы вполне можете
    ' ее опустить и написать следующую строку как Set rdorst=
    'rdConn.OpenResultset("MyTable",rdOpenDynamic,rdConcurValues).
    'Мы просто хотим указать, что при обращении
    'в методе OpenResultset к
    'объекту rdoTable требуется его имя.
    'Впрочем, об этом сказано в Справке.
    Set rdoTab = rdConn.rdoTables("MyTable")
    'Инициализация переменной набора данных
    Set rdoRst = rdConn.OpenResultset(rdoTab.Name, _
    rdOpenDynamic, rdConcurValues)
    'Инициализация переменной колонки
    Set rdoCol = rdoRst.rdoColumns(1)
    rdoRst.Edit
    rdoCol.Value = 27
    rdoRst.Update
    Me!text1 = rdoCol.Value
    Exit Sub
rdflInitializeErr:
    If Err.Number = rdoEngine.rdoErrors(0).Number And _
        rdoEngine.rdoErrors.Count >> 1 Then

```

```

For Each errX In rdoEngine.rdoErrors
    MsgBox "Error " & errX.Number & " вызвана " _
        & errX.Source & ": " & errX.Description, _
        vbCritical, "rdfInitialize()"
Next errX
Else
    MsgBox "Error " & Err.Number & " вызвана " _
        & Err.Source & ": " & Err.Description, _
        vbCritical, "rdfInitialize()"
End If
End Sub

```

Объект `rdoResultset` можно создать с помощью метода `OpenResultSet`, который применяется к объектам `rdoConnection`, `rdoTable` и `rdoPreparedStatement` (о последнем объекте речь впереди). В силу этого метод `OpenResultSet` имеет два вида синтаксиса, один из которых относится к объекту `rdoConnection`

```

Set Variable = Connection.OpenResultSet(Source[,
Type[, Locktype[, Options]])

```

другой к объектам `rdoTable` и `rdoPreparedStatement`

```

Set Variable = Object.OpenResultSet([ Type[,
Locktype [, Options]])

```

Пример первого варианта синтаксиса можно найти в предыдущей процедуре

```

Set rdoRst = rdConn.OpenResultSet(rdoTab.Name, _
rdOpenDynamic, rdConcurValues)

```

Первым аргументом является источник данных, которым может быть объект `rdoTable`, как показано в предыдущем примере, объект `rdoPreparedStatement` или SQL выражение.

Таким образом, мы можем создать совершенно одинаковый набор данных, используя различные варианты синтаксиса и объекты в качестве аргументов. Например, используем SQL выражение:

```

Set rdoRst=rdConn.OpenResultSet("SELECT * FROM MYTABLE", _
rdOpenDynamic, rdConcurValues)

```

При использовании второго варианта синтаксиса аргумент *Source* (источник данных) отсутствует, так как объект, к которому вы применяете метод, сам по себе является источником данных.

```

Set rdoTab = rdConn.rdoTables("MyTable")
Set rdorst=rdoTab.OpenResultSet(rdOpenDynamic,rdConcurValues)

```

Если сравнить два последних фрагмента, становится видно, что первый вариант более гибок, так как, добавив условие с помощью ключевого слова `WHERE`, мы можем значительно сократить выборку и соответственно значительно уменьшить нагрузку на ресурсы, выиграв при этом в скорости выполнения.

Теперь самое время поговорить об объекте `rdoPreparedStatement`, после чего мы продолжим разговор о методе `OpenResultSet`.

Объект `rdoPreparedStatement` создается с помощью метода объекта `rdoConnection` `CreatePreparedStatement`, который имеет следующий синтаксис:

```

Set Prepstmt = Connection.CreatePreparedStatement(Name, Sqlstring)

```

Аргумент *Name* - это имя вновь создаваемого объекта `rdoPreparedStatement`. Следующий аргумент - *Sqlstring* - является правильным выражением SQL. Оба аргумента обязательны, но их можно заменить пустой строкой (""). Объект, созданный с помощью этого метода, автоматически добавляется к коллекции `rdoPreparedStatement`s. При этом если вы не снабдили его именем, то есть использовали пустую строку, то обращаться к нему можно с помощью переменной объекта `prepstmt` или по порядковому номеру в коллекции - `rdoPreparedStatement`s(2).

У вновь созданного объекта `rdoPreparedStatement` имеется коллекция `rdoParameters`, с помощью которой вы можете передавать параметры в аргумент *Source* метода `OpenResultSet`. Например:

```
SQLStr="SELECT * FROM Account WHERE account = ? AND _summa >> ?"
Set myprepst=myrdConn. CreatePreparedStatement("FromAccount",SQLStr)
'Далее мы можем подставить значения параметров
myprepst.rdoParameters(0)=104
myprepst.rdoParameters(1)=25000
Set myrdRst=myprepst._ OpenResultSet(rdOpenDynamic,rdConcurValues)
```

Вернемся к методу `OpenResultSet`, а точнее к его аргументу *Type*, который может иметь одно из четырех значений типа `Integer`.

Значение	Получаемый тип набора данных
rdOpen-ForwardOnly	Объект <code>rdoResultSet</code> , в котором поиск записей может производиться только сверху вниз (от первой к последней записи). Указатель записи нельзя вернуть назад к первой записи, одновременно доступна только одна запись. Наборы данных такого типа используются для быстрого выбора и обработки данных
rdOpenStatic	Порядок и значения в наборе данных статического курсора фиксируются при его открытии. Изменения, добавления и удаления, произведенные другими пользователями, не будут появляться до закрытия и последующего открытия курсора.
RdOpen-Keyset	Результат запроса может иметь изменяемые строки. Допускается перемещаться между записями. Этот набор данных вы можете использовать, чтобы добавлять, изменять или удалять данные из соответствующей таблицы или таблиц. Членство этого набора данных фиксировано.
RdOpen-Dynamic	Результат запроса может иметь изменяемые строки. Допускается перемещаться между записями. Этот набор данных вы можете использовать, чтобы добавлять, изменять или удалять данные из соответствующей таблицы или таблиц. Членство этого набора данных не фиксировано.

Выбирайте значение аргумента в зависимости от требуемой функциональности. Если вам не нужно редактировать данные на сервере, то имеет смысл выбирать значение `rdOpenForwardOnly`. То есть если вы строите отчет за какой-то период, то вам нет смысла отслеживать изменения, которые произойдут в течение следующего промежутка времени. Если же программа помогает продавать железнодорожные билеты, то ей постоянно нужны свежие данные о количестве свободных мест. Наиболее надежным в этом плане является динамический тип набора данных, но с ним и тяжелее всего работать.

Следующий аргумент *LockType* служит для контроля за блокировками страниц, которые содержат редактируемую или записываемую на диск запись. Может принимать значение типа `Integer`, равное одной из следующих констант:

Значение	Тип разрешения проблем одновременного доступа
rdConcurLock	Пессимистический тип, то есть страница блокируется, как только вы выполнили метод <code>Edit</code> . В это время другие пользователи редактировать данные на этой странице не могут.
RdConcur-	Курсор открывается только для

ReadOnly	чтения, следовательно, никакие блокировки не требуются
rdConcur-Rowver	Оптимистическая блокировка, то есть запись блокируется только во время выполнения метода Update . При этом поиск записи в исходном наборе происходит по идентификатору записи.
rdConcurValues	Оптимистическая блокировка, то есть запись блокируется только во время выполнения метода Update . При этом поиск записи в исходном наборе происходит по значениям строки.

Последний аргумент **Option** служит для установки асинхронного режима выборки данных. При этом режиме программа может продолжать выполняться, не ожидая завершения выборки всех данных.

Последними объектами в иерархии **RDO** является коллекция колонок - **rdoColumns**. С помощью объекта **rdoColumn** вы получаете доступ непосредственно к значениям таблиц в конкретных строках. При этом для перемещения по записям кроме оговоренных выше методов используются методы **MoveFirst**, **MoveLast**, **MoveNext** и **MovePrevious**, очень похожие на те, которыми вы пользовались при программировании с помощью **DAO**.

8.6. Внешнее управление сервером с помощью SQL-DMO

Данный параграф относится только к **Microsoft SQL Server**. В нем мы опишем методы управления сервером с помощью пользовательского приложения.

В клиентской части настольной базой данных может использоваться приложение, которое способно выступать в качестве **OLE** контроллера. Тем, кто прочитает [десятую главу](#) данной книги или хорошо знает **Visual Basic**, будет очень легко освоиться с набором объектов, который предоставляет **DMO**. Для того чтобы использовать данную технологию с рабочей станции, необходимо установить на рабочей станции клиентскую версию **Microsoft SQL Server**. Установить ее очень легко. Например, при установке **SQL Server** на рабочей станции, которая работает под управлением **Windows 95**, программа установки сама определит тип операционной системы и, соответственно, предложит для использования на данном компьютере клиентскую часть сервера. При этом вам будут доступны многие компоненты **SQL Server**, например **Enterprise Manager** или **Books On-line**.

В любом случае для работы с **SQL-DMO** на ПЭВМ, работающей под управлением **Windows 95** или **Windows NT**, должны быть установлены следующие файлы:

- **SQLOLE.HLP** - файл контекстной помощи для работы с **SQL-DMO**;
- **SQLOLE.REG** - файл регистра;
- **SQLOLE65.DLL** - in-process сервер **SQL-DMO** и программные компоненты;
- **SQLOLE65.TLB** - библиотека объектов, используемая программой контроллером **OLE Automation**;
- **SQLOLE65.SQL** - файл поддержки языка **transact-SQL** для создания хранимых процедур.

DMO - **Distributed Management Object** - это распределенные объекты управления. Главное назначение данной технологии - управление сервером с любой рабочей станции в сети.

DMO может использоваться и для выполнения запросов. Правда, при этом вы не можете получить для использования в своем приложении курсоры и, соответственно, у вас нет средств навигации по таблицам. Запросы могут быть запросами выборки и запросами действия, или, иначе, запросами определения данных.

Несмотря на то, что мы не можем получить удобный курсор с методами для навигации и отображения данных, которые бы хранились в нем, использовать **SQL-DMO** для доступа к конкретным данным можно. Для этого необходимо использовать объект **QueryResults**, который отображает данные запроса выборки и имеет методы для работы с колонками и строками.

На рис. 8.15 приведена иерархия объектов **SQL-DMO** и их коллекций, а в табл. 8.8 содержится краткое описание этих объектов.

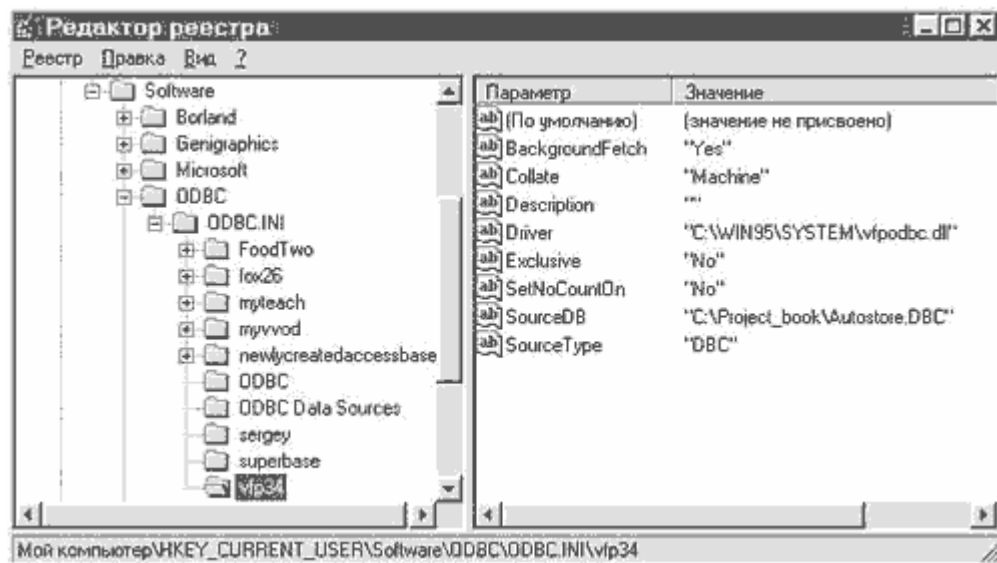


Рис. 8.15.

Таблица 8.8. Назначение объектов SQL-DMO

Объект	Описание
Alert	Содержит всю информацию, которая описывает предупреждения SQL Server, появляющиеся при наступлении определенных событий.
AlertSystem	Управляет процессом выполнения предупреждений SQL Server.
Application	Обеспечивает доступ и управление средой приложения, в том числе коллекцией объектов SQL Server, которые связаны с используемыми в системе отдельными серверами.
Article	Содержит информацию о статье, помещенной в публикацию. Статья - это таблица, представление или их часть, синхронизируемые по содержанию на внешнем и локальном серверах. С помощью свойств TableName и DestinationTableName задаются таблицы или представления, участвующие в синхронизации. Свойства InsertCommand, UpdateCommand и DeleteCommand позволяют задать операторы transact-SQL, которые будут выполняться при вставке, обновлении или удалении данных. Свойства ReplicateAllColumns и ReplicationFilterProcName позволяют ограничить по колонкам или строкам данные, помещаемые в статью.
Backup	Используется для выполнения резервного копирования или восстановления БД или журнала транзакций.
BulkCopy	Содержит информацию для копирования данных из таблиц или представлений в файл данных (текстовый файл) или из такого файла в таблицу SQL Server. Позволяет установить границы копирования и разделительные символы. Для копирования используются методы ExportData или ImportData объектов Table или View.
Check	Содержит информацию об ограничениях целостности, которые могут содержать одну или более колонок. С помощью свойства ExcludeReplication эти ограничения могут отключаться во время синхронизации данных.
Column	Позволяет получить информацию об имеющихся в таблице колонках, типе хранящихся в них данных,

	значениях по умолчанию, допустимости значений NULL и т. д.
Configuration	Позволяет получить информацию о конфигурации сервера. Включает только коллекцию объектов ConfigValue , каждый из которых имеет минимальное, максимальное и текущее значение. После изменения параметров конфигурации получить новые значения можно с помощью метода ReconfigureCurrentValue или ReconfigureWithOverride .
ConfigValue	Сохраняет минимальное, максимальное и текущее значение параметров конфигурации сервера.
Database	Контейнер базы данных, который используется для ссылки на все содержащиеся в данной БД объекты, такие как Table , StoredProcedure и т. д. Служит также для выполнения запросов с помощью метода ExecuteWithResult .
DBObject	Содержит информацию, применимую ко всем объектам БД и позволяет использовать один и тот же код для управления несколькими объектами, такими как Table , View , Rule и т. д.
DBOption	Содержит все опции для БД, которые может установить ее владелец. Например, возможность подписки на данные из этой БД, права доступа и т. п.
Default	Значение по умолчанию, хранящееся для колонки или типа данных.
Device	Предоставляет возможность получить информацию о имеющихся устройствах сервера (имя, физическое расположение, объем и т. п.). С помощью этого объекта вы можете полностью управлять процессом зеркальной записи данных и изменениями, выполняемыми параллельно на двух физических дисках.
DRIDefault	Содержит ограничения целостности по умолчанию для колонок.
Executive	Содержит информацию о выполнении плановых задач и предупреждений.
Group	Содержит информацию о группе пользователей.
History-Filter	Используется для определения или очистки списка заданий с помощью методов EnumHistory и PurgeHistory объектов SQLServer и Task .
Index	Индекс для указанной таблицы. Позволяет получить информацию об имеющихся индексах, перестроить индекс, удалить, создать новый или скопировать существующий индекс для другой таблицы.
IntegratedSecurity	Определяет, как система доступа и обеспечения секретности SQL Server будет интегрироваться с аналогичными системами для бюджетов пользователей и групп пользователей Windows NT .
Key	Содержит информацию о первичном, уникальном или внешнем ключах таблицы. Каждый ключ может включать одну или более колонок таблицы.
Language	Содержит полную информацию о каждом из языков, поддерживаемых SQL Server .
Login	Идентификатор пользователя и пароль для соединения с сервером. Этот объект имеет глобальный характер в целом для SQL Server ; ему соответствует объект User в каждой БД.
Names	Эта коллекция служит для хранения имен

	объектов в виде строк символов.
Operator	Описывает условия работы оператора, в том числе его почтовый адрес, необходимость получения предупреждений, режим работы и т. д.
Permission	Содержит информацию о пользовательских объектах БД или привилегиях команд.
Publication	Содержит информацию о публикации (наборе статей), используемой для синхронизации данных. Метод <code>AddFullSubscriber</code> позволяет указать сервер, на котором данные синхронизируются по всем статьям в наборе.
QueryResults	Содержит данные, полученные при выполнении запроса. Данные могут быть получены путем выполнения метода <code>ExecuteWithResults</code> объектов <code>SQLServer</code> , <code>Database</code> и <code>RemoteServer</code> . Данные размещаются в памяти и не могут редактироваться.
RegisteredServer	Содержит имя и данные о регистрации SQL серверов, входящих в объект <code>ServerGroup</code> для использования в <code>SQL Enterprise Manager</code> . Этот объект отображается в <code>Server Manager</code> и запоминается в Регистре.
Registry	Содержит всю информацию, размещенную в Регистре <code>Windows</code> о <code>SQL Server</code> .
Remote-Login	Используется при соединении внешнего сервера с локальным <code>SQL Server</code> и содержит необходимую для этого соединения информацию.
Remote-Server	Внешний <code>SQL Server</code> , который становится известным для локального сервера.
Rule	Содержит информацию о допустимых данных для колонок или типов данных.
Server-Group	Группа серверов, имеющая уникальное имя.
SQLServer	Объект для связи с используемым сервером. Каждый объект <code>SQLServer</code> может инициализировать одно соединение с выбранным сервером, используя метод <code>Connect</code> .
Stored-Procedure	Поименованный набор команд <code>Transact-SQL</code> , который запоминается в БД. Помимо команд в свойстве <code>Text</code> может храниться имя и путь к библиотеке <code>DLL</code> , реализующей расширенный вариант хранимых процедур.
Subscriber-Info	Содержит всю необходимую информацию для обновления данных на внешнем сервере, включая дату начала и конца подписки (свойства <code>ActiveStartDate</code> и <code>ActiveEndDate</code>), интервалы ежедневного, еженедельного и ежемесячного обновления и т. д.
Subscription	Содержит информацию, необходимую для синхронизации данных.
System-Datatype	Содержит информацию о системных типах данных <code>SQL Server</code> .
Table	Представляет набор строк в БД (таблицу). Позволяет изменять свойства таблицы, добавлять новые колонки, проверять ограничения целостности и т. д.
Task	Содержит информацию о плановых заданиях.
TransactionLog	Содержит информацию о процедуре регистрации транзакций. Позволяет зарезервировать пространство для журнала регистрации и указать необходимость его ведения на отдельном устройстве.

Transfer	Содержит информацию для пересылки данных и объектов из БД источника (объект Database) в другую БД, расположенную на внешнем сервере.
Trigger	Триггер для указанной таблицы. Позволяет получить информацию о хранящихся триггерах или создать новый с помощью метода Script. Тип триггера определяется свойством Type.
User	Содержит информацию о пользователе БД. Каждый пользователь ассоциируется с одной регистрацией пользователя для работы с БД.
User-Defined-Datatype	Определяет пользовательский тип данных.
View	Представление, хранящееся в БД. В свойстве Text хранится команда SELECT Transact-SQL для выполнения запроса. Данные в представлении не могут редактироваться.

На вершине иерархии объектов DMO находится объект Application, который обеспечивает нас методами и свойствами для управления средой приложения.

Важной особенностью использования объектов SQL-DMO является наличие наряду с самими объектами, их коллекций. За счет этого работа с объектами в Visual Basic существенно упрощается. Коллекция включает группу объектов одного типа. Имя коллекции образуется как множественное число от имени объекта. К коллекции можно обратиться из объекта, являющегося родительским в иерархии. Если объект не имеет коллекции, то используется коллекция Properties, которая имеет по одному объекту Property для каждого свойства. С ее помощью можно легко установить имена и значения всех свойств объекта, не зная их заранее. Например:

```
For Each oProperty in oSQLServer.Properties
    Debug.Print oProperty.Name & " = " oProperty.Value
Next
```

В Visual FoxPro возможность использования таких языковых конструкций появится только в следующей версии. При использовании версии 3.0 работы немного прибавится. Например, для получения всех имен БД, имеющихся на сервере, в Visual Basic достаточно такого фрагмента кода:

```
For Each oDatabase in oSQLServer.Databases
    Debug.Print oDatabase.Name
Next
В Visual FoxPro код будет чуть длиннее:
nDBNum = oSQLServer.Databases.Count
FOR nItem = 1 TO nDBNum
    ? oSQLServer.oDatabases(nItem).Name
NEXT
```

Для добавления нового объекта в коллекцию используется метод Add. Для удаления объекта из коллекции - метод Remove.

По сравнению с другими рассмотренными ранее объектными моделями, модель объектов SQL-DMO имеет особый тип коллекции, - список (list), доступный только для чтения. Возможности получения списков весьма широки и охватывают почти все объекты DMO, но необходимо учитывать, что в списке вы получите данные, фиксирующие текущее состояние системы и никак не изменяющиеся при дальнейших действиях.

Для того чтобы начать работу с SQL сервером посредством DMO, если вы используете Access или Visual Basic, необходимо создать объект SQLServer:

```
Global MyServer As New SQLOLE.SQLServer
```

Если вы используете Visual FoxPro, то необходимо воспользоваться функцией CREATEOBJECT():

```
oMyServer = CREATEOBJECT("SQLOLE.SQLServer")
```

В данный момент этот объект представляет собой абстрактную ссылку. Для придания ему конкретности необходимо подсоединиться к существующему серверу, естественно, если вам как пользователю даны права или вы знаете пароль администратора.

Вы можете сделать это, используя метод `Connect` объекта `SQLServer`, например, следующим образом:

```
oMyServer.Connect(Servername:="Autoserver",_ Login="Me",Password="You")
```

Можно пойти более сложным путем, но при этом вы будете знать причину возникновения возможной ошибки. Используйте следующую или подобную ей функцию:

```
Function SQLLogin(strServerName As String, _strUID As String, strPWD As String) As Boolean
' Подключение к указанному SQL серверу, используя
' предложенный идентификатор пользователя и пароль.
' Устанавливает глобальный объект типа SQLServer
' Возвращает истину при удачном завершении,
' в противном случае - ложь.
On Error GoTo SQLLoginErr
Const erSQLAlreadyLoggedIn = -2147211004
SQLLogin = True
DoCmd.Hourglass True
mySQLServer.Connect strServerName, strUID, _strPWD
SQLLoginExit:
DoCmd.Hourglass False
Exit Function
SQLLoginErr:
Select Case Err.Number
Case erSQLAlreadyLoggedIn
SQLLogin = True
Resume Next
Case Else
MsgBox "Error " & Err.Number & ": " & _
Err.Description, vbCritical, "SQLLogin()"
End Select
SQLLogin = False
Resume SQLLoginExit
End Function
```

В простейших случаях можно безболезненно пользоваться первой конструкцией. Необходимо помнить, что после завершения работы программы с использованием SQL-DMO вам обязательно надо отсоединиться от сервера. Для этого используйте метод `Disconnect` объекта `SQLServer`:

```
oMySQLServer.Disconnect
```

В промежутках между присоединением к серверу и отсоединением от него вам доступны практически любые операции с сервером и с его данными, но в зависимости от предоставленных вам прав доступа. Вы можете, например, получить информацию о доступных устройствах баз данных, о самих базах данных, выполнять запросы и работать с результатами запросов.

Если нужно создать новую базу данных, которая будет располагаться на конкретном устройстве, то можно это сделать примерно так, как это сделано в следующем примере:

```
Function SQLMakeNewDatabase(strName As String, _strDeviceName As String, lngSize As Long)
' Создаем новую БД на текущем SQL Server
On Error GoTo SQLMakeNewDatabaseErr
SQLMakeNewDatabase = True
Dim objNewDatabase As New SQLOLE.Database
If Not gobjSQLServer Is Nothing Then
DoCmd.Hourglass True
With objNewDatabase
.Name = strName
End With
' БД будет располагаться на устройстве по
' умолчанию
gobjSQLServer.Databases.Add objNewDatabase
```

```

        ' Увеличиваем размер указанного устройства
        With objNewDatabase
            .ExtendOnDevices (strDeviceName & "=" & _ CStr(IngSize))
            .Shrink (IngSize)
        End With
    Else
        SQLMakeNewDatabase = False
    End If
SQLMakeNewDatabaseExit:
    DoCmd.Hourglass False
    Exit Function
SQLMakeNewDatabaseErr:
    MsgBox "Error " & Err.Number & ": " & _Err.Description, _vbCritical, "SQLMakeNewDatabase()"
    SQLMakeNewDatabase = False
    Resume SQLMakeNewDatabaseExit
End Function

```

Следующий фрагмент кода показывает, как можно добавить новую колонку во все существующие таблицы.

```

Dim oTable As Table
Dim oCol As Column
Set db = ss.Databases("pubs")
For Each oTable In db.Tables
    ' Выполняем для каждой таблицы в коллекции
    Set oCol = New Column
    oCol.AllowNulls = True
    oCol.Datatype = "varchar"
    oCol.Length = 30
    oCol.Name = "NewCol"
    oTable.BeginAlter ' Начинаем процесс изменения
    oTable.InsertColumn oCol, ""
    oTable.DoAlter ' Завершаем процесс изменения
Next oTable

```

Вероятно, вы заметили, что перед добавлением колонки мы применяем метод **BeginAlter**, а для выполнения всех изменений необходимо воспользоваться методом **DoAlter**. При этом на сервере реально будут совершены все действия, записанные после применения метода **BeginAlter**. Этот процесс напоминает транзакции и должен использоваться при выполнении действий со следующими объектами:

- Alert
- AlertSystem
- Article
- Executive
- Operator
- Publication
- RemoteServer
- SubscriberInfo
- Subscription
- Table
- Task

Это условие вызвано тем, что при выполнении любых действий с перечисленными объектами должны приниматься повышенные меры безопасности, и, в частности, должна быть возможна безболезненная отмена этих действий. Для других объектов изменение значений свойств и выполнение каких-либо методов приводят к непосредственному их обновлению.

В заключение отметим, что одно из наиболее перспективных направлений использования SQL-DMO - проведение процесса перевода настольных приложений в технологию клиент-сервер.

Глава 9

Разработка пользовательского интерфейса

9.1. Инструментарий разработчика

9.2. Конструируем форму

Создание формы "Прием заказов" на Visual FoxPro

Создание формы "Прием заказов" на Access

9.3. Разработка управляющего меню

Разработка меню в Visual FoxPro

Разработка меню в Access

Даже программисты быстро забыли, что еще несколько лет назад общались с компьютером с помощью колоды перфокарт, а результат работы программы могли видеть лишь на листе бумаги. Теперь самую лучшую программу никто не оценит по достоинству, если она не будет иметь удобного пользовательского интерфейса.

При работе с современными средствами разработки приложений программист с самого начала окунается в проблемы построения интерфейса. Также и мы, обсуждая проблемы работы с данными, не могли не коснуться этих проблем. В этой главе мы как бы подведем итог обсуждаемым ранее вопросам под привычным лозунгом Microsoft "Putting It All Together!" - "Объединим все вместе!"

Рассуждая о принципах и методах построения пользовательского интерфейса можно много говорить о различных писанных и неписанных правилах. А можно этого не делать, а попросить читателя повнимательнее посмотреть на тот интерфейс, который обеспечивает любимое средство разработки. Посмотрели? Это лучший образец для подражания.

9.1. Инструментарий разработчика

В [третьей главе](#) мы уже рассматривали визуальный инструментарий для разработки пользовательского интерфейса. Все три пакета для построения пользовательских программ (Visual FoxPro, Visual Basic и Access) имеют соответствующие средства, выполненные в виде Конструкторов и Мастеров. Если в Visual Basic и Access основным визуальным инструментарием для построения пользовательского интерфейса следует считать Конструктор формы, то в Visual FoxPro сюда следует отнести и Конструктор класса, который может эффективно использоваться для разработки многократно используемых элементов от отдельных объектов до целых форм.

В этом параграфе мы рассмотрим средства, доступные для разработчика пользовательского приложения программным путем.

В первую очередь это, конечно, средства языка программирования. С помощью программного кода мы можем описывать действия, которые приложение должно выполнять при наступлении событий, подобных выбору пользователем какого-либо пункта меню или щелчку мыши на том или ином элементе пользовательского интерфейса. Возможности языков программирования Visual FoxPro и Visual Basic в этом плане достаточно широки, примерно равны по мощности и могут обеспечить самую изощренную функциональность для вашей программы. Авторы выражают слабую надежду, что читатель успел в этом убедиться, прочитав предыдущие главы нашей книги. Поэтому более подробно мы остановимся на дополнительных средствах, доступных программисту в рассматриваемых средствах разработки.

И все-таки даже такие мощные программные средства порой бессильны перед жесткими требованиями пользователя и неудержимым желанием программиста сделать свою программу самой совершенной в мире. В этом случае на помощь можно призвать дополнительные средства в виде богатого набора функций операционной системы - Windows API.

Windows API - это интерфейс, позволяющий прикладной программе использовать функции, встроенные в операционную систему.

При использовании функций Windows API программист может реализовать следующие преимущества:

- Повышение скорости работы. В большинстве случаев функции, встроенные в ОС, обеспечивают более высокую скорость выполнения операций, чем встроенные функции Visual FoxPro или Visual Basic. В первую очередь это касается выполнения графических операций.

- Стабильность и высокая надежность работы функций, связанная с высокой степенью отладки ОС.
- Отсутствие потребности в дополнительных модулях. Альтернативой использования Windows API является приобретение дополнительных компонентов, обеспечивающих расширенную функциональность, что повышает затраты на разработку пользовательского приложения и создает дополнительные сложности при его распространении.

В то же время мы рекомендуем использовать Windows API только в крайнем случае и при необходимости повышения скорости работы программы, так как здесь программисту приходится сталкиваться с более сложными программными конструкциями. Это, естественно, требует времени на освоение, так как те сложные задачи, которые были скрыты от программиста при использовании любимого языка программирования высокого уровня с приставкой Visual, теперь придется решать самому. При этом учтите, что Microsoft до сих не выпустила хорошего руководства или четко структурированного файла контекстной помощи по Windows API и вам придется долго блуждать среди многих сотен функций с мудреными названиями. С профессиональными версиями Visual FoxPro (только на CD-ROM) и Visual Basic поставляется файл со справочными сведениями по Windows API, но он содержит синтаксис для программирования на языке Си.

Если вы все же не раздумали использовать функции Windows API, в своей программе вам придется позаботиться о двух необходимых составляющих. Во-первых, вы должны определить путь для передачи параметров в вызываемую функцию. Во-вторых - определить механизм для получения значения, возвращаемого функцией. Эти задачи выполняет команда DECLARE.

В Visual Basic она имеет следующий синтаксис:

```
[Public | Private] Declare Function FunctionName Lib "LibraryName" [Alias "AliasName" ]
[([ParamList])][As ParamType]
```

В Visual FoxPro при том же составе параметров синтаксис команды выглядит чуть иначе:

```
DECLARE [cFunctionType] FunctionName IN LibraryName [AS AliasName] [cParamType1 [@]
ParamName1,cParamType2 [@] ParamName2, ...]
```

С помощью этой команды можно зарегистрировать функцию, расположенную в динамической библиотеке и затем использовать ее как стандартную. В команде выделим четыре основных компонента:

- Сведения о функции определяются ее именем *FunctionName*, заданном в библиотеке DLL. Имя функции следует указывать с соблюдением регистра! Если функция возвращает какое-либо значение, перед ее именем следует указать тип возвращаемого значения *cFunctionType*. Вероятно, здесь уместно напомнить, что в Visual Basic объявление Public используется для обеспечения доступа к функции из всех процедур, расположенных во всех модулях. Объявление Private обеспечивает доступ к функции только внутри данного модуля.
- Имя библиотеки DLL Windows *LibraryName*, содержащей требуемую функцию. Если вы хотите использовать функцию, входящую в API Windows, то здесь достаточно написать WIN32API, что автоматически обеспечит поиск функции в одном из следующих файлов: KERNEL32.DLL, GDI32.DLL, USER32.DLL, MPR.DLL или ADVAPI32.DLL.
- Псевдоним имени функции для использования в Visual FoxPro или в Visual Basic. Его удобно использовать для сокращения слишком длинного названия функции или при угрозе совпадения имени функции с зарезервированным словом.
- Список передаваемых параметров включает обозначение типа и имени параметров, передаваемых из приложения в функцию DLL Windows. Знак @ после типа параметра показывает, что он будет передаваться по ссылке, а не по значению. Имя параметра никак не используется ни приложением, ни функцией DLL и может применяться в команде только в информационных целях. Если функция DLL требует в качестве аргумента передачи нулевого указателя (null pointer), то его следует передавать как целое число по значению.

Для передачи параметров в Visual Basic следует использовать следующий синтаксис:

```
[Optional][ByVal | ByRef][ParamArray] varname[( )][As ParamType]
```

Опции **ByVal** или **ByRef** определяют, что параметр будет передан по значению или по ссылке. В следующей таблице показано, к каким результатам приведет передача параметров по значению или по ссылке различного типа:

Тип параметра	По значению	По ссылке
Integer	Помещает в стек вызовов 16-битовое значение.	Помещает в стек вызовов 32-битовый адрес. Этот адрес обеспечивает ссылку на 16-битовое значение параметра.
Long	Помещает в стек вызовов 32-битовое значение. Этот адрес обеспечивает ссылку на 32-битовое значение параметра.	Помещает 32-битовый адрес в стек.
String	Преобразует строку символов в формат, принятый в C, с нулевым (CHR(0)) символом в конце. Помещает 32-битовый адрес этой строки в стек.	Помещает 32-битовый указатель в стек.

Помимо правильного использования параметров, довольно важным при использовании функций DLL является применение указателя.

Указатель (**handle**) - это число, уникально идентифицирующее объект.

Среда Windows поддерживает очень много объектов различного типа. Это разнообразные окна, блоки памяти, меню, шрифты, палитры и т. д. Допустим, вы хотите нарисовать голубую линию; тогда вы должны создать объект "голубой карандаш". Функция Windows API **CreatePen()** вернет указатель на этот "карандаш", который вы затем можете использовать для рисования. Когда вы закончите работу, этот объект можно убрать, передав указатель в функцию **DestroyObject()**.

Рассмотрим несколько примеров для демонстрации возможностей Windows API. Начнем с таких задач, которые невозможно выполнить стандартными средствами.

Давайте посмотрим, как сделать в Visual FoxPro список, в котором будут появляться все устройства, доступные на данном компьютере. При этом для дисководов гибких дисков должна проверяться их готовность к работе (наличие дискеты). Этот список лучше всего оформить в виде класса. Тогда при добавлении его в форму мы сразу получим требуемый элемент управления.

Список доступных логических устройств для данного компьютера мы можем получить с помощью следующей функции API Windows (в синтаксисе C, как приведено в файле WIN32API.HLP):

DWORD GetLogicalDriveStrings (DWORD *nBufferLength*, LPTSTR *lpBuffer*);

Эта функция возвращает список доступных устройств в виде строки символов. Тип устройства можно определить с помощью функции

UINT GetDriveType (LPCTSTR *lpRootPathName*);

Эта функция возвращает одно из следующих значений:

- 0 - тип устройства не определен;

- 1 - нет загрузочного сектора;
- 2 - устройство для сменного носителя данных;
- 3 - жесткий диск;
- 4 - сетевое устройство;
- 5 - дисковод для лазерных дисков;
- 6 - виртуальный диск.

Эта функция поможет нам украсить список устройств соответствующим его типу изображением.

Для определения готовности дисковода к записи нам потребуется еще одна функция, которая переопределяет реакцию операционной системы на критические ошибки, например, на отсутствие дискеты в дисковом дисководе при попытке записи:

UINT SetErrorMode(UINT fuErrorMode);

Эта функция возвращает предыдущую установку реакции на ошибки. В Visual FoxPro нет такого многообразия типов данных, как в Си ++, поэтому тип возвращаемого этими тремя функциями значения **DWORD** (двойное слово) или **UINT** (целое без знака) заменим имеющим 4 бита **INTEGER**.

Теперь, проведя необходимую теоретическую подготовку, можно приступить к практическому созданию требуемого класса.

В Project Manager при активной вкладке **Classes** щелчком на кнопке **New** и в диалоговом окне **New Class** укажем название создаваемого класса, класс, на основании которого он создается, и имя библиотеки, где он будет храниться. Будем надеяться, что к этому моменту у вас уже есть собственный набор классов и класс для списка в том числе.

Дадим этому классу имя **IstDrives** и установим для него следующие свойства:

- **BoundColumn** = 2 (две колонки);
- **ColumnLines** = .F. (без разграничительных линий);
- **FontSize** = 12 (размер шрифта, обычно лучше использовать 10, но отдельные буквы можно увеличить).

Код для создания требуемого списка в событии **Init** может выглядеть следующим образом:

```
* Определяем функцию для получения списка доступных
* устройств
DECLARE INTEGER GetLogicalDriveStrings IN Win32API AS GetDrive;
INTEGER, STRING
* Определяем функцию для определения типа устройства
DECLARE INTEGER GetDriveType IN Win32API AS GetType;
STRING
* Описываем локальные переменные
LOCAL cLetter, nLetter, NType, nDrivesNum
* Резервируем строку для размещения списка доступных
* устройств
IpString=SPACE(200)
* Определяем размер буфера для строки
nBuffSize=LEN(IpString)
* Получаем список устройств
= GetDrive(nBuffSize,@IpString)
* Определяем количество устройств
nDrivesNum = OCCURS(":",IpString)
* Получаем их имена
FOR nLetter = 1 TO nDrivesNum
  cLetter = UPPER(" " + SUBSTR(IpString,AT(":",IpString,nLetter)-1,1))
  * Определяем тип устройства
  nType = GetType(ALLTRIM(cLetter + ":\\"))
  * Добавляем обозначение устройства (первая колонка)
  * и его тип (вторая колонка) в список
  This.AddItem(cLetter, nLetter, 1)
  This.List(nLetter, 2) = STR(nType)
  * Задаем соответствующее изображение
  DO CASE
    CASE NType = 2
```



```

        This.Picture(nLetter) = "FLOPPY.BMP"
    CASE NType = 3
        This.Picture(nLetter) = "DRIVE.BMP"
    CASE NType = 4
        This.Picture(nLetter) = "NET.BMP"
    CASE NType = 5
        This.Picture(nLetter) = "CDROM.BMP"
    ENDCASE
ENDFOR

```

Все необходимые файлы изображений вы найдете на дискете. Нам их пришлось создать заново, так как обширная библиотека изображений, поставляемая с **Visual FoxPro**, содержала подходящие изображения только в формате ICO 32x32. В подобных списках лучше смотрятся изображения 16x16. Впрочем, это легко сделать с помощью утилиты **Imagedit** из состава **Visual FoxPro**. Здесь только необходимо учесть, что при выводе изображения **Visual FoxPro** очень своеобразно обходится с белым цветом, считая, что его вообще нет. Если вы выведете изображение, содержащее белый цвет, например, на серую кнопку, то те места изображения, которые были белые, станут серыми. Для сохранения белого цвета для каждого изображения надо создать двойника - маску (файл с расширением **MSK**), в котором на месте белого цвета должен располагаться черный. В качестве примера этой операции можно руководствоваться изображениями, которые использует **Wizard** (Мастер) для графических кнопок управления в форме.

В событии **Click** класса запишем следующий код:

```

LOCAL cLetter
cLetter = This.DisplayValue + ":"
* Если устройством является дисковод для гибких дисков,
* то проверяем наличие
* дискеты с помощью метода IsDiskIn
IF ALLTRIM(This.Value) = "2"
    IF This.IsDiskIn(cLetter) = .T.
        This.Parent.cmdOk.Enabled = .T.
    ELSE
        = MESSAGEBOX("Вставьте дискету в выбранное устройство!",0,;
            "Не готов дисковод")
    IF This.IsDiskIn(cLetter) = .T.
        This.Parent.cmdOk.Enabled = .T.
    ELSE
        This.Parent.cmdOk.Enabled = .F.
    ENDIF
ENDIF
RETURN
This.Parent.cmdOk.Enabled = .T.

```

Для определения наличия дискеты в дисковом диске создадим в нашем классе путем выбора команды **New Method** в меню **Class** специальный метод **IsDiskIn()** и запишем в него следующий код:

```

LPARAMETERS cDrive
LOCAL IOldError
* Регистрируем функцию SetErrorMode, которая определяет,
* как ОС реагирует на некоторые критические ошибки,
* в том числе дисковые операции
DECLARE INTEGER SetErrorMode IN Win32API INTEGER
* Определяем наличие и сохраняем имя обработчика ошибок
IOldError = ON('ERROR')
ON ERROR IDiskError = .T.
* Задание функции SetErrorMode с аргументом 1 позволяет
* приложению самому обрабатывать критические ошибки
* вместо ОС
nOldErrorState = SetErrorMode(1)
* По умолчанию считаем, что ошибки нет, если не так,
* значение будет изменено
IDiskError = .f.

```

```

* Пытаемся найти NUL файл в загрузочной области
* указанного устройства
IDriveState = FILE(cDrive + "\NUL")
IF .NOT. IDiskError
    IF IDriveState
        IDriveOk = .t.
    ELSE
        IDriveOk = .f.
    ENDIF
ELSE
    IDriveOk = .f.
ENDIF
* Восстанавливаем прежний обработчик ошибок
IF .NOT. EMPTY(IOldError)
    ON ERROR DO (IOldError)
ELSE
    ON ERROR
ENDIF
* Восстанавливаем реакцию на ошибки ОС
nRestState = SetErrorMode(nOldErrorState)
RETURN IDriveOk

```

После этого класс списка можно помещать в любой требуемой форме.

Во втором примере использования функций Windows API рассмотрим возможность ускорения выполнения графических операций.

Сначала создадим форму, в которой будем выводить зигзагообразную линию средствами Visual FoxPro (используя метод Line). Для этого в методе Paint формы запишем следующий код:

```

nY = 1
FOR nX = 5 TO 375 STEP 5
    ThisForm.Line(nX, nY*100)
    nY = -nY
ENDFOR

```

Теперь попробуем такую же линию нарисовать средствами Windows API. В методе Paint второй формы запишем:

```

*** Возвращает указатель на контекст указанного окна
DECLARE INTEGER GetDC IN WIN32API INTEGER
* HWND Указатель окна
*** Стирает контекст указанного окна
DECLARE INTEGER ReleaseDC IN WIN32API INTEGER, INTEGER
* HWND Указатель окна
* HDC Указатель контекста устройства
*** Рисует линию с текущей позиции до указанных координат
DECLARE INTEGER LineTo IN WIN32API INTEGER, INTEGER, INTEGER
* HDC Указатель контекста устройства
* int nXEnd x-координата конечной точки линии
* int nYEnd y-координата конечной точки линии
*** Обновляет текущее положение указанной точки
DECLARE INTEGER MoveToEx IN WIN32API INTEGER, INTEGER, INTEGER, STRING
* HDC Указатель контекста устройства
* int X x-координата нового текущего положения
* int Y y-координата нового текущего положения
* LPPOINT адрес предыдущего положения (NULL in FoxPro)
*** Функция GetFocus не имеет параметров и возвращает HWND активного окна
DECLARE INTEGER GetFocus IN WIN32API
mynull = .NULL.
pmhand = GetFocus()
IF pmhand<<>>0 && Если есть открытое активное окно
    hmdc = GetDC(pmhand)
    = MoveToEx(hmdc, 0, 0, mynull)
    nY = 1
    FOR nX = 5 TO 375 STEP 5
        = LineTo(hmdc, nX, nY*100)
    
```

```

        nY = -nY
    ENDFOR
    = ReleaseDC(pmhand, hmdc)
ELSE
    WAIT WINDOW "Нет активного окна"
ENDIF

```

Запустите обе формы и вы убедитесь, что второй вариант работает в несколько раз быстрее. Для рисования линии средствами Windows API нам пришлось использовать четыре функции. Передаваемые параметры перечислены в строчках комментария после объявления функций.

В этом примере мы используем новое понятие - контекст устройства (device context). Что это такое?

Контекст устройства - это совокупность параметров, описывающих условия работы с каким-либо устройством.

Использование контекста устройства позволяет программисту сосредоточиться на выполнении какого-то действия (например, на написании кода для рисования) и избежать имеющих вспомогательное значение операций. Для рисования такими вспомогательными операциями могут быть установленное разрешение экрана, число поддерживаемых данным устройством цветов и т. д. Если вы пишете код для рисования красного прямоугольника, то можете быть уверены, что Windows обеспечит рисование этого объекта с учетом возможностей того устройства, которое выбрано для его отображения, будь то экран, принтер или плоттер.

С точки зрения программиста такой подход обозначает рисование в объекте, называемом контекстом устройства. При этом контекст устройства может рассматриваться как "черный ящик", обладающий определенными характеристиками. Важнейшие из них:

- Система координат, которая может быть представлена такими фиксированными единицами, как сантиметры, дюймы или пиксели, или комбинированными единицами.
- Параметры рисуемой линии (объект Pen) определяют цвет, ширину и стиль линии.
- Параметры закрашивания (объект Brush) определяют цвета, используемые для заполнения фона, и палитру для графических команд.
- Шрифт (объект Font) описывает шрифт, используемый при работе команд вывода данных.

В заключение обсуждения проблем, связанных с использованием функций Windows API, затронем еще один важный вопрос. Он связан с тем, что многие функции Windows API в качестве параметров используют структуры.

При использовании Visual Basic у программиста не появится никаких проблем - это средство разработки поддерживает структуры. Visual FoxPro не поддерживает структур и программисту придется формировать структуру в виде символьной строки. В качестве примера такой работы приведем фрагмент программного кода для соединения с сервером по телефонной линии. Само соединение предварительно создается в специальной утилите внешнего доступа Dial-Up Networking. Windows хранит параметры созданного соединения в регистре. В Visual FoxPro требуемый фрагмент будет выглядеть так:

```

DECLARE INTEGER RasDial IN rasapi32 ;
    STRING DialExtentions, ;
    STRING cPhonebookfile, ;
    STRING @ cParameters, ;
    INTEGER nCallBack, ;
    STRING cCallBack, ;
    STRING @ nConnHandle
nConnHandle = CHR(0)+CHR(0)+CHR(0)+CHR(0)
&& Указатель соединения
cConnName = "RcomPPP" && Имя соединения
cPhone = "2594277"
&& Номер телефона, подключенного к серверу
cUserName = "Administrator" && Имя пользователя
cPassWord = "AndreyG" && Пароль для соединения
cDomain = "" && Имя домена Windows NT
* Формируем структуру
cParam = CHR(28)+CHR(4)+CHR(0)+CHR(0)+;

```

```

PADR(cConnName+CHR(0), 256+1)+;
PADR(CHR(0), 128+1)+;
PADR(CHR(0), 128+1)+;
PADR(cUserName+CHR(0), 256+1)+;
PADR(cPassword+CHR(0), 256+1)+;
PADR(cDomain+CHR(0), 15+1)+" "
* Выполняем соединение
nRes = RasDial(NULL, NULL, @cParam, 0, NULL, @nConnHandle)

```

Равный по функциональности фрагмент, написанный на Visual Basic, будет выглядеть так:

```

Private nConnHandle As Long, nRes As Long
' Описываем структуру параметров соединения
Private Type DIALPARAMS
    dwSize As Long
    szEntryName(256 + 1) As String
    szPhoneNumber(128 + 1) As String
    szCallbackNumber(128 + 1) As String
    szUserName(256 + 1) As String
    szPassword(256 + 1) As String
    szDomain(15 + 1) As String
End Type
Private ConnParams As DIALPARAMS
' Функция для установки соединения
Private Declare Function RasDialA Lib "rasapi32" _
    (DialExtentions, _
    cPhonebookfile, _
    ByRef Parameters As DIALPARAMS, _
    nCallBack As Long, _
    cCallBack, _
    ByRef hConnHandle As Long) As Long
` Присваиваем значения параметрам
nRes = 0
nConnHandle = 0
` Это значение dwSize устанавливается при внешнем доступе из Windows 95
ConnParams.dwSize = 1052
' Наименование соединения в регистре Windows
ConnParams.szEntryName(0) = "R"
ConnParams.szEntryName(1) = "c"
ConnParams.szEntryName(2) = "o"
ConnParams.szEntryName(3) = "m"
ConnParams.szEntryName(4) = "P"
ConnParams.szEntryName(5) = "P"
ConnParams.szEntryName(6) = "P"
ConnParams.szEntryName(7) = Chr(0)
' Номер телефона берем из регистра
ConnParams.szPhoneNumber(0) = Chr(0)
ConnParams.szCallbackNumber(0) = Chr(0)
ConnParams.szUserName(0) = "A"
ConnParams.szUserName(1) = Chr(0)
ConnParams.szPassword(0) = Chr(0)
ConnParams.szDomain(0) = Chr(0)
` Выполняем соединение
nRes = RasDialA(Null, Null, ConnParams, 0, Null, nConnHandle)

```

9.2. Конструируем форму

При описании различных действий, возникающих при работе с данными, мы уже многократно говорили о формах, которые составляют основу интерфейса любого приложения для обработки данных.

В этом параграфе мы более детально разберем процесс проектирования форм для работы с данными при использовании различных СУБД.

Формы предназначены для представления, ввода и редактирования данных удобным и привычным для пользователя способом. Помимо этого в формы можно включать различные

объекты, реагирующие на события (системные или пользовательские). С помощью этих объектов можно разрабатывать простые и интуитивно понятные интерфейсы пользователя для управления данными.

Несмотря на то, что вы полностью можете описать форму с помощью программирования, как правило, более простым и быстрым способом будет визуальный. При этом Конструктор формы возьмет на себя труд перевода ваших пожеланий на соответствующий язык программирования. Язык, который используется в Visual Basic, а значит, и в Access, в отличие от Visual FoxPro не является объектно-ориентированным, тем не менее он объектный. Таким образом, какой бы путь создания формы вы ни выбрали, работа будет заключаться в создании объектов и обеспечении требуемой функциональности за счет использования их свойств и методов. Большинство свойств позволяют указать действия, которые должны выполняться при наступлении событий, доступных для конкретного объекта. Используя Конструктор формы, с помощью окна Свойства (Properties) мы получаем легкий доступ к любому из свойств и можем описать действия объекта в ответ на события. В коде событий мы можем использовать методы объектов, например, для перемещения на него курсора (придания статуса активного объекта в форме).

Например, в форме "Прием заказов" мы будем использовать командные кнопки, комбинированный список и текстовое поле. Для переноса элементов управления на проектируемую форму можно использовать панель инструментов с элементами управления. Для установки некоторых визуальных свойств формы, например цвета фона и шрифта, можно использовать панель инструментов для форматирования. При этом очень часто использованию панелей инструментов есть хорошая альтернатива в виде всплывающего меню, вызываемого при нажатии правой кнопки мыши и являющееся контекстным по отношению к выделенному элементу.

Формы при работе с данными используют свой набор данных. Организация процесса взаимодействия формы с данными происходит посредством установки свойства RecordSource (Источник данных) и связанных с конкретными полями источника данных объектов. Если вы хотите отображать с помощью элементов управления данные из полей, которые не относятся к источнику данных, то прямое указание в свойстве ControlSource (Данные) ни к чему не приведет. Вам необходимо организовать синхронизацию записей между данными, входящими в источник данных, и данными, не входящими, и, естественно, использовать несвязанное поле для отображения информации, не относящейся к источнику данных. Если вы ничего не поняли из этой фразы, то правильно сделали. Как правило, так не поступают, потому что подобный путь требует сложного и не оправдывающего себя кодирования. Лучший способ - это создание запроса, который одновременно может отобразить данные из нескольких таблиц. Помимо этого используйте подчиненные формы. При этом необходимо иметь поля, по которым вы можете связать данные в основной и подчиненной формах. Еще проще вызывать форму, содержащую нужные вам, но хранящиеся в другом источнике данные, динамически, с помощью реакции на события. Во многих случаях этого более чем достаточно.

Иногда, при решении более сложных задач, когда вам необходимо, чтобы никакие обстоятельства, вплоть до перебоев в энергоснабжении, не смогли нарушить установленные правила взаимодействий между данными, вам необходимо использовать транзакции. Для использования транзакций в Access необходимо работать с такими наборами данных, создание и доступ к которым осуществляются с помощью методов объектов доступа к данным (DAO).

Опишем форму, которую мы будем создавать в качестве примера.

На основе таблицы Order_ создается форма для принятия заказа. Ни одно из полей в таблице Order_ пользователь непосредственно не заполняет. Путем использования комбинированных списков и вызываемых форм он заполняет другие таблицы, при этом автоматически заполняются поля в новой записи таблицы Order_. Перед началом редактирования новой записи начинается транзакция.

Поле key_order заполняется автоматически, так как имеет тип Счетчик. Поле key_salman заполняется именем текущего пользователя приложения (для примера на Access) или выбирается из списка продавцов (для примера на Visual FoxPro). Заказчик выбирается из списка заказчиков по фамилии, имени и адресу или заносится вновь (для примера на Access), при этом необходимо вызвать форму для заполнения данных о заказчике. При вызове формы для заполнения данных о заказчике также начнется транзакция, но не вложенная, а параллельная, так как данные о заказчике будут сохраняться в любом случае, для того чтобы потом "засыпать" его проспектами и приглашениями на презентации новых моделей автомобилей.

После этого, на основе данных, хранящихся в таблице Model, подбирается модель и, если она есть на складе и удовлетворяет заказчика, то заполняется запись в таблице Account. После этого завершается транзакция и запись вносится в таблицу физически.

Создание формы "Прием заказов" на Visual FoxPro

Начнем с систематизации данных о способах создания новой формы в Visual FoxPro. Таких способов достаточно много:

- Использование Мастера форм обсуждалось ранее. Это эффективный способ создания простых форм или "заготовок" для более сложных форм.
- Выбор команды *New* из меню *File* главного меню Visual FoxPro. В открывшемся диалоговом окне *New* необходимо щелкнуть на кнопке *Form*.
- Выполнение команды **CREATE FORM**.
- В *Project Manager* выделение пункта *Form* и выбор *New*.

Независимо от выбранного способа загружается Конструктор формы, окно которого с основными визуальными инструментами представлено на рис. 9.1. При этом в главном меню Visual FoxPro появляется меню *Form*, с помощью которого можно выполнить следующие действия:

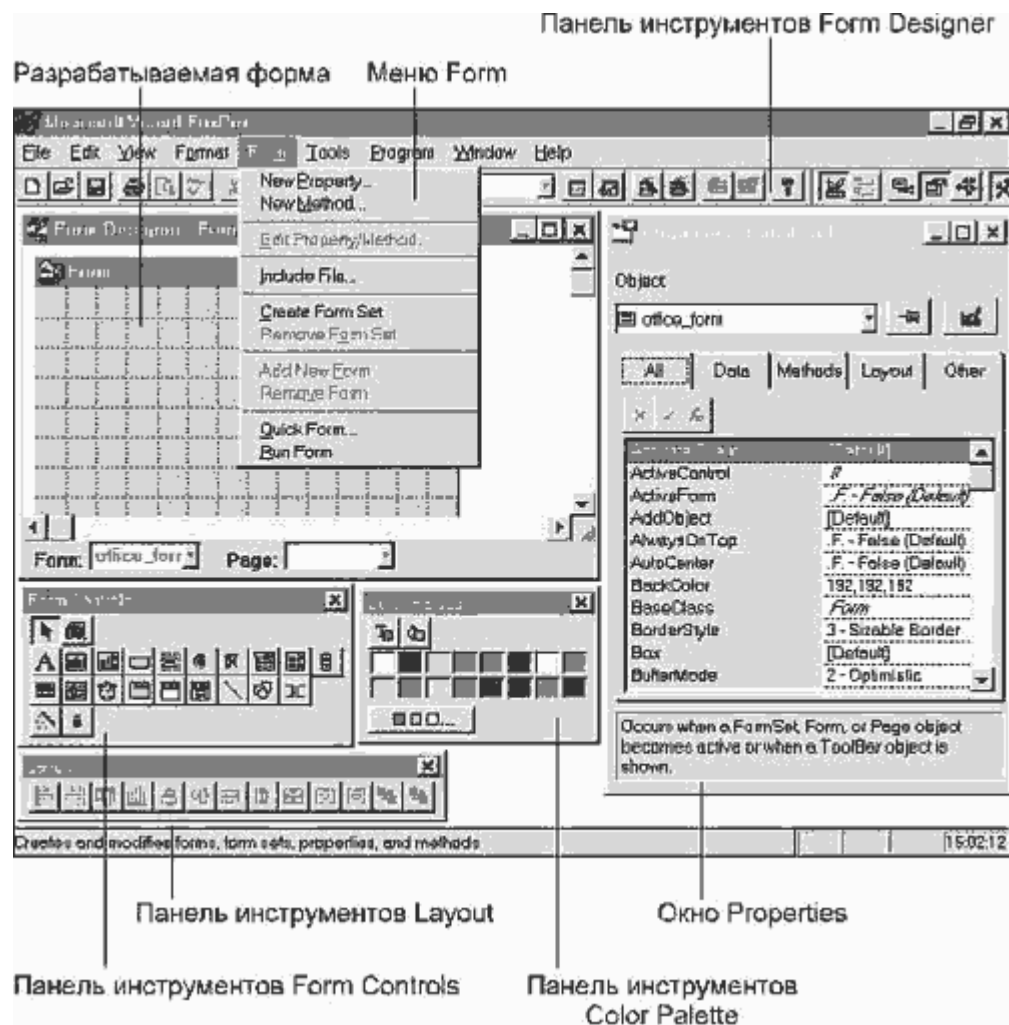


Рис. 9.1.

- *New property* - позволяет добавить свое свойство для создаваемой формы.
- *New method* - позволяет добавить свой метод для создаваемой формы.
- *Edit Property/Method* - позволяет удалить добавленные свойства и методы из описания формы или изменить комментарий.
- *Include File* - позволяет определить файл, содержащий директивы компилятора.
- *Create Form Set* - создает набор форм - объект, который будет являться контейнером по отношению к нескольким формам и панелям инструментов.
- *Remove Form Set* - удаляет набор форм, если он создан.
- *Add New Form* - позволяет добавить форму в набор форм.
- *Remove Form* - удаляет форму из набора форм.
- *Quick Form* - запускает **Построитель формы** (Form Builder), с помощью которого можно быстро создать простую форму для работы с данными из одной таблицы и в последующем использовать как заготовку в Конструкторе формы. Построитель формы представляет собой упрощенную версию Мастера формы.
- *Run Form* - запускает разрабатываемую форму.

Для ускорения разработки формы служат несколько панелей инструментов.

Панель инструментов Form Designer предназначена для быстрого перехода или запуска различных элементов Конструктора формы. Она появляется автоматически при запуске Конструктора формы. Назначение кнопок на этой панели описано на рис. 9.2.



Рис. 9.2.

Панель инструментов Layout предназначена для быстрого изменения расположения объектов на завершающем этапе создания формы. Ее возможности будут вам ясны, если вы посмотрите на рис. 9.3. Перед выполнением какого-либо действия предварительно нужно выбрать объект или группу объектов, удерживая клавишу **Shift** и щелкая на них мышкой. Операции выравнивания, которые выполняются относительно группы объектов, за образец берут самый большой размер, иначе перед выполнением операции выравнивания необходимо удерживать клавишу **Ctrl**. Если выбранное действие не дало желаемого результата или вовсе привело к нежелательным последствиям, не стоит отчаиваться, вспомните про кнопку **Undo** на стандартной панели инструментов Visual FoxPro.

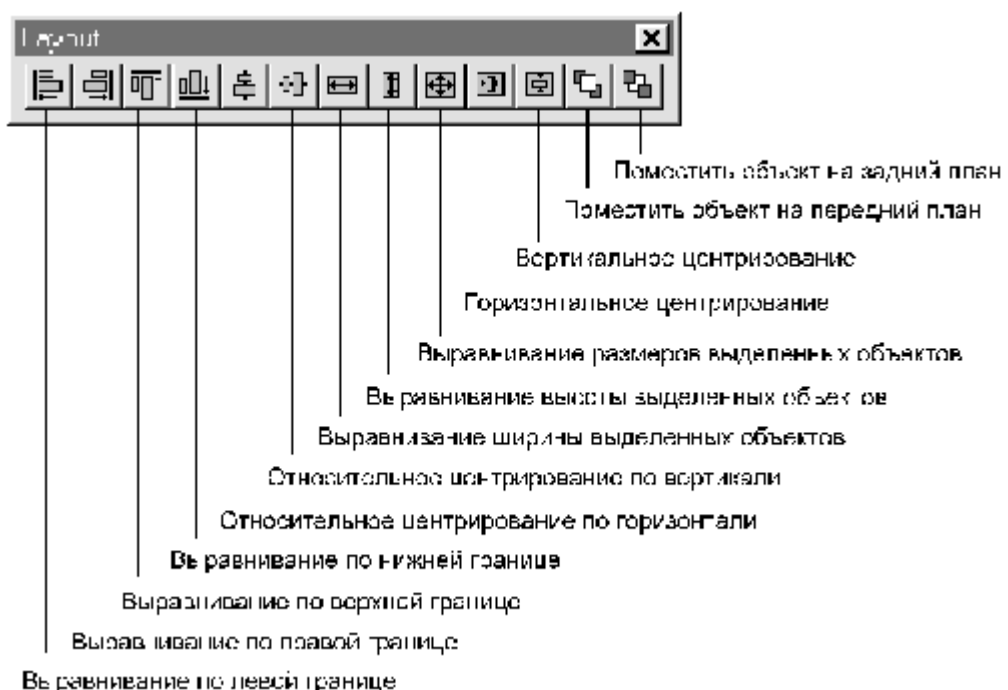


Рис. 9.3. Панель инструментов Layout

Панель инструментов Form Controls позволяет включать в разрабатываемую или редактируемую форму тот или иной элемент управления. На рис. 9.4 описаны кнопки данной панели инструментов.

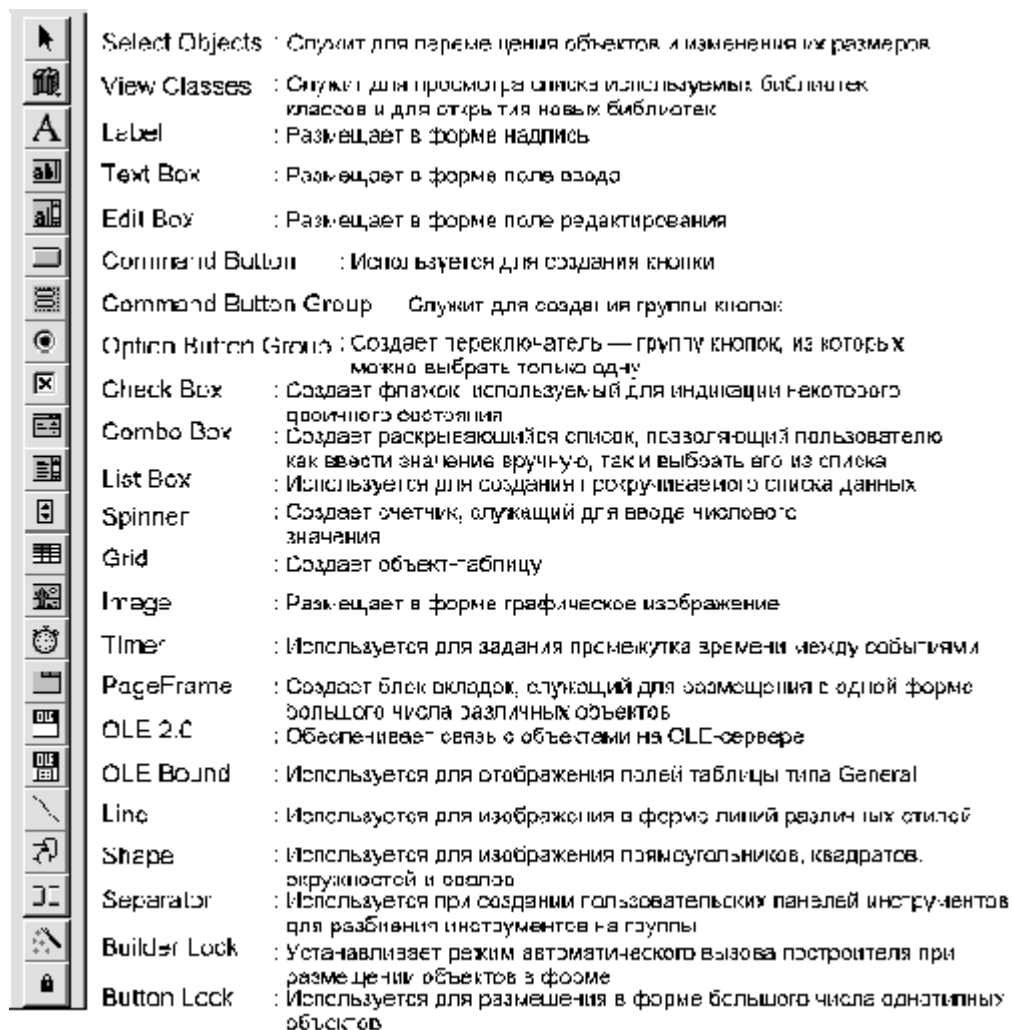


Рис. 9.4.

Перед тем как вы начнете разрабатывать формы для своего приложения, не поленитесь заглянуть в диалоговое окно **Options** меню *Tools*. Выберите вкладку *Forms* и установите на ней требуемые значения для расположенных там параметров (рис. 9.5). Для облегчения расположения элементов управления на форме включите вывод координатной сетки. Наиболее удобный шаг координатной сетки - от 6 до 10. В качестве единицы измерения установите пиксели - *Pixels*. В опции *Maximum Design Area* установите разрешение 640x480 - это означает, что разработанная вами форма наверняка поместится на экране, имеющем стандартное разрешение *VGA*. Если вы установите более высокое разрешение, то форма может прекрасно выглядеть на вашем компьютере, но окажется слишком большой на экране компьютера у пользователя вашей программы.



Рис. 9.5.

Если какие-либо свойства проектируемого объекта по умолчанию имеют отличные от требуемых значения, то нужные значения свойств можно установить в окне **Properties**, которое для удобства поиска требуемого свойства или метода разбито на пять вкладок: "All", "Data", "Methods", "Layout", "Other". Элементы окна **Properties** показаны на рис. 9.6.

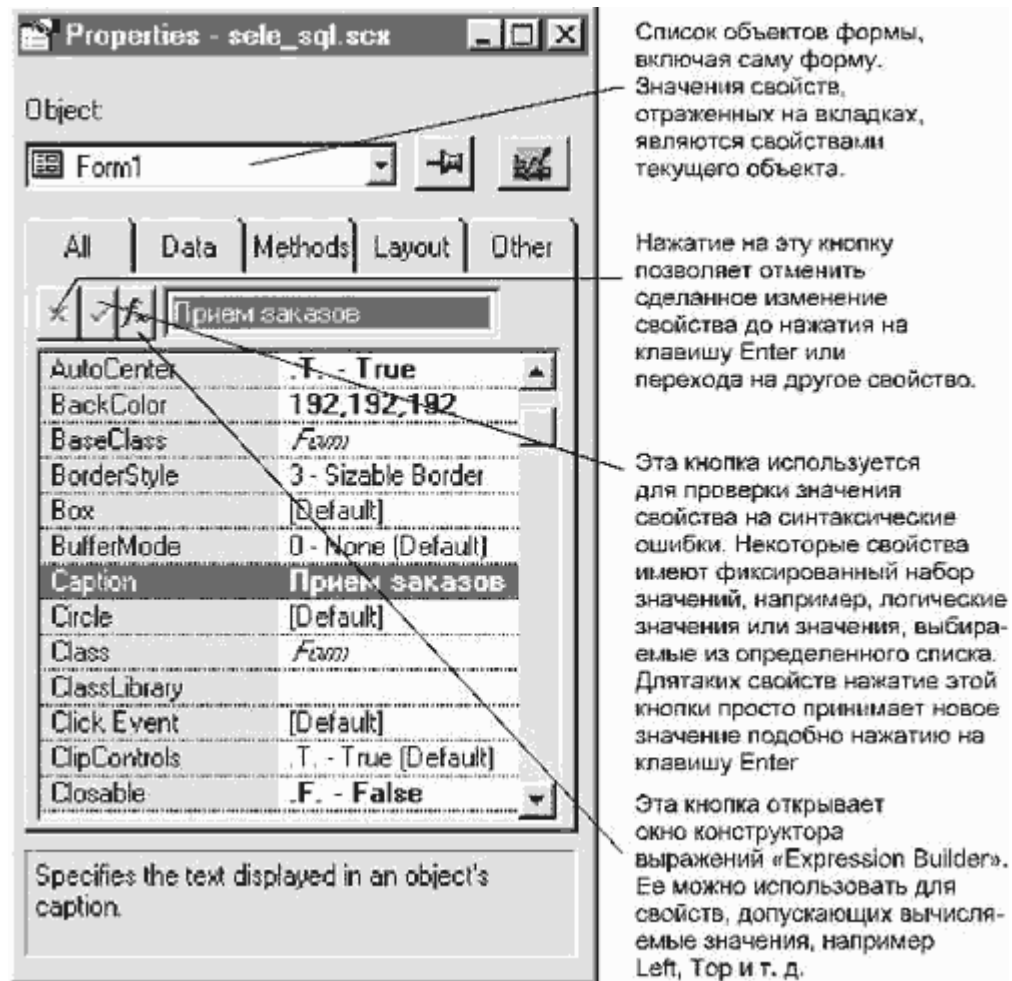


Рис. 9.6. Список свойств в Конструкторе формы (окно Properties)

Вкладка "All" содержит все элементы, включенные в остальные разделы окна.

Убедимся с помощью окна Properties, что форма имеет следующие свойства:

AutoCenter = .T. - определяем автоматическое центрирование объекта Form;
 BackColor = RGB(176,176,176) - задаем цвет фона;
 Caption = Прием заказов - задаем текст, отображаемый в названии объекта;
 Closable = .F. - ликвидируем возможность закрытия объекта Form, двойным щелчком кнопки управляющего меню, или выбора в этом меню команды Close;
 Height = 300 - задаем ширину объекта на экране;
 Icon = "c:\mybook\sample\net13.ico" - задаем значок, который отображается на этапе выполнения для объекта Form при его сворачивании;
 MaxButton = .F. - ликвидируем доступ к кнопке Maximize;
 MinButton = .F. - ликвидируем доступ к кнопке Minimize;
 ShowTips = .T. - определяем возможность вывода подсказки для элементов управления заданного объекта;
 Width = 540 - задаем ширину объекта.

Следующим этапом мы включаем необходимые элементы управления, такие как Label, Text Box, Command Button, Combo Box, Shape, и определяем уже для вновь созданных элементов их свойства.

Для управления работой формы создадим специальный элемент управления. Скорее всего мы будем использовать его и в других формах. В этом случае самым лучшим решением будет создание нового класса *mygo* на базе класса *Container* (контейнера). После этого мы сможем легко включить в нашу форму собственный объект *mygo1*, который основан на базе класса *mygo*. Прервем ненадолго описание процесса создания формы, чтобы уделить необходимое внимание новому классу.

При создании классов в Конструкторе класса необходимо определить базовый класс для нового класса и задать имя библиотеки, в которую этот класс будет записан. Для создания класса необходимо выбрать команду *New* из меню *File* главного меню Visual FoxPro. В открывшемся диалоговом окне *New* щелкаем на кнопке выбора *Class* и нажимаем кнопку *New File*. В окне *New Class* вводим имя класса (*Class Name*), выбираем имя класса, на котором будет

основан создаваемый класс (**Based On**), и имя библиотеки (**Store In**), в которой будет храниться новый класс (рис. 9.7). Либо в Project Manager выбрать вкладку Classes и нажать кнопку New.

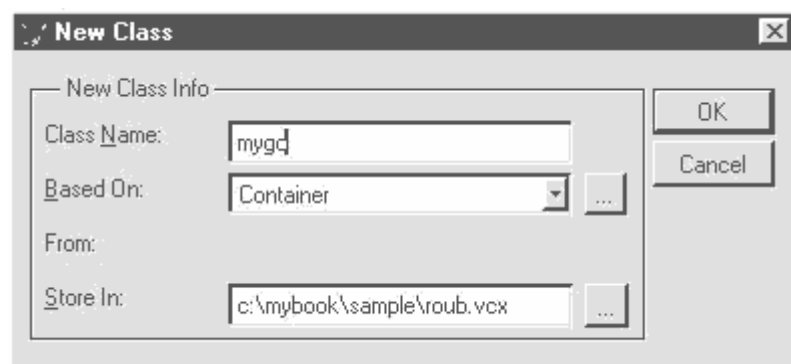


Рис. 9.7. Диалоговое окно New Class

Как видно из рис. 9.7, имя нашего класса **mygo**, базовый класс **Container** и имя библиотеки, в которую будет включен класс, - **Roub.vcx**

Вы можете включать в форму объекты своего класса прямо из панели инструментов Form Controls, если предварительно зарегистрируете класс.

Зарегистрировать библиотеку классов в Конструкторе формы можно следующим образом:

1. Нажмите кнопку **View Classes** панели инструментов **Form Controls**.
2. В открывшемся меню выберите команду **Add**.
3. В диалоговом окне **Open** укажите библиотеку классов и нажмите кнопку **Open**.

Класс **mygo** содержит кнопки для перемещения указателя записи на первую запись таблицы, на предыдущую запись таблицы, на следующую запись таблицы, на последнюю запись таблицы, а также кнопки поиска, добавления новой записи, удаления текущей записи, сохранения изменений, отмены изменений и кнопку выхода из формы. На рис. 9.8 показан внешний вид класса **mygo**. В табл. 9.1 приведены имена элементов управления в классе **mygo** и значения свойства **ToolTipText** для каждого объекта.



Рис. 9.8. Разработка нового класса в Конструкторе класса

Таблица 9.1. Назначение элементов класса **mygo**

Объект	Значение
Com1	Первая запись
Com2	Назад
Com3	Поиск
Com4	Вперед
Com5	Последняя запись
Com6	Добавить
Com7	Удалить
Com8	Сохранить
Com9	Отменить
Com10	Выход из формы

Добавляем последовательно два новых метода для нашего класса, `my_show_s_u` и `myrefresh`, выбрав в меню *Form* команду *New Method*.

В код метода `my_show_s_u` запишем следующие строчки:

```
ThisForm.mygo1.Com8.Enabled = .F.
ThisForm.mygo1.Com9.Enabled = .F.
MyUpdate = 0
IF Del_ = 0
    ThisForm.mygo1.Myrefresh
ELSE
    Del_ = 0
ENDIF
```

В код метода `myrefresh` запишем:

```
ThisForm.Text1.Refresh
ThisForm.Text2.Refresh
ThisForm.Text3.Refresh
ThisForm.Text4.Refresh
ThisForm.Refresh_list
```

В код события Click для кнопки `Com1` запишем:

```
GO TOP
ThisForm.mygo1.Com1.Enabled=.F.
ThisForm.mygo1.Com2.Enabled=.F.
ThisForm.mygo1.Com4.Enabled=.T.
ThisForm.mygo1.Com5.Enabled=.T.
ThisForm.mygo1.MyRefresh
```

В код события Click для кнопки `Com2` запишем:

```
IF BOF()=.F.
SKIP -1
    ThisForm.mygo1.Com4.Enabled=.T.
    ThisForm.mygo1.Com5.Enabled=.T.
ELSE
    ThisForm.mygo1.Com1.Enabled=.F.
    ThisForm.mygo1.Com2.Enabled=.F.
    GO TOP
ENDIF
```

```
ThisForm.mygo1.MyRefresh
```

В код события Click для кнопки `Com3` запишем:

```
DO FORM Find_ord.scx
```

В код события Click для кнопки `Com4` запишем:

```
IF EOF()=.F.
ThisForm.mygo1.Com1.Enabled=.T.
ThisForm.mygo1.Com2.Enabled=.T.
SKIP 1
    IF EOF()=.T.
        ThisForm.mygo1.Com4.Enabled=.F.
        ThisForm.mygo1.Com5.Enabled=.F.
        GO BOTTOM
    ENDIF
ELSE
ThisForm.mygo1.Com4.Enabled=.F.
ThisForm.mygo1.Com5.Enabled=.F.
GO BOTTOM
ENDIF
```

```
ThisForm.mygo1.MyRefresh
```

В код события Click для кнопки `Com5` запишем:

```
GO BOTTOM
ThisForm.mygo1.Com1.Enabled=.T.
ThisForm.mygo1.Com2.Enabled=.T.
ThisForm.mygo1.Com4.Enabled=.F.
ThisForm.mygo1.Com5.Enabled=.F.
ThisForm.mygo1.MyRefresh
```

В код события Click для кнопки `Com6` запишем:

```
INSERT INTO order_view (key_salman, key_customer, key_model) ;
VALUES (order_view.key_salman, order_view.key_customer, order_view.key_model)
MyUpdate=1
ThisForm.mygo1.Com8.Enabled=.T.
```

```

ThisForm.mygo1.Com9.Enabled=.T.
ThisForm.Text1.Visible=.F.
ThisForm.mygo1.MyRefresh
    В код события Click для кнопки Com7 запишем:
Answer_d = MESSAGEBOX("Удалить данную запись ?", 4+32+256, "Вопрос")
IF Answer_d=6
    DELETE
    Del_=1
    SKIP 1
    IF EOF()=.T.
        GO BOTTOM
    ENDIF
    MyUpdate=1
    ThisForm.mygo1.Com8.Enabled=.T.
    ThisForm.mygo1.Com9.Enabled=.T.
    ThisForm.mygo1.MyRefresh
ENDIF
    В код события Click для кнопки Com8 запишем:
=TABLEUPDATE(.T.,.T.)
ThisForm.mygo1.My_Show_s_u
    В код события Click для кнопки Com9 запишем:
=TABLEREVERT(.T.)
ThisForm.mygo1.My_Show_s_u
    В код события Click для кнопки Com10 запишем:
IF MyUpdate=1
    Answer_s_u = MESSAGEBOX("Сохранить изменения ?", 4+32, "Вопрос")
    DO CASE
        CASE Answer_s_u=6 &&Да
            =TABLEUPDATE(.T.,.T.)
        CASE Answer_s_u=7 &&Нет
            =TABLEREVERT(.T.)
    ENDCASE
    ThisForm.mygo1.My_Show_s_u
ENDIF
ThisForm.Release

```

Далее приведен программный код для событий проектируемой формы. На рис. 9.9 показан внешний вид разрабатываемой формы и отношения между описанными ниже объектами, событиями и методами.

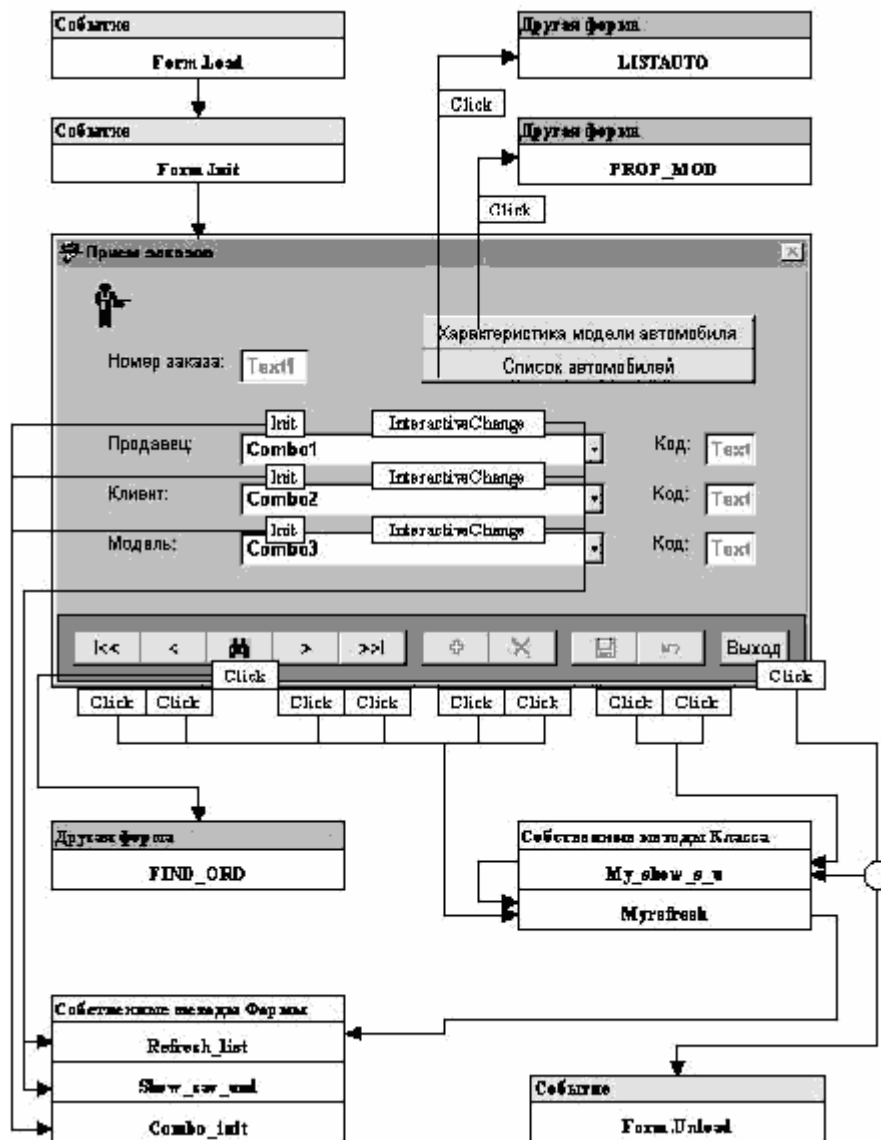


Рис. 9.9. Отношения между объектами, событиями и методами в разрабатываемой форме

Код для события Load формы:
 && В администраторе ODBC необходимо иметь соединение auto к БД Auto_store
 OPEN DATABASE Auto_store EXCLUSIVE
 CREATE CONNECTION remote_01 DATASOURCE auto
 DO CASE
 CASE LevDostup=1
 =DBSETPROP('remote_01', 'CONNECTION', 'ConnectionString', ;
 'DSN=auto;UID=login_lev1;PWD=lev1')
 CASE LevDostup=2
 =DBSETPROP('remote_01', 'CONNECTION', 'ConnectionString', ;
 'DSN=auto;UID=login_lev2;PWD=lev2')
 CASE LevDostup=3
 =DBSETPROP('remote_01', 'CONNECTION', 'ConnectionString', ;
 'DSN=auto;UID=login_lev3;PWD=lev3')
 CASE LevDostup=4
 =DBSETPROP('remote_01', 'CONNECTION', 'ConnectionString', ;
 'DSN=auto;UID=login_lev4;PWD=lev4')
 CASE LevDostup=5
 =DBSETPROP('remote_01', 'CONNECTION', 'ConnectionString', ;
 'DSN=auto;UID=login_lev5;PWD=lev5')
 ENDCASE
 && Создаем просмотры
 a_v=ADBOBJECTS(a_view, 'VIEW')
 IF a_v>0
 prn=ASCAN(a_view, 'propmodel_view')
 lau=ASCAN(a_view, 'lauto_view')

```

smn=ASCAN(a_view, 'sman_view')
cus=ASCAN(a_view, 'cust_view')
mod=ASCAN(a_view, 'model_view')
ord=ASCAN(a_view, 'order_view')
ENDIF
IF prn=0
CREATE SQL VIEW propmodel_view CONNECTION remote_01 SHARE ;
AS SELECT model.key_model, model.name_model, model.swept_volume, ;
model.quantity_drum, model.capacity, model.torgue, model.top_speed, model.starting, ;
model.quantity_door, model.quantity_sead, model.length, model.width, ;
model.height, model.expense_90, model.expense_120, model.expense_town, ;
firm.name_firm, country.name_country, fuel_oil.name_fuel_oil, ;
tyre.name_tyre, body.name_body ;
FROM model ,firm, country, fuel_oil, tyre, body ;
WHERE model.key_firm=firm.key_firm ;
AND firm.key_country=country.key_country ;
AND model.key_fuel_oil=fuel_oil.key_fuel_oil ;
AND model.key_tyre=tyre.key_tyre ;
AND model.key_body=body.key_body
ENDIF
IF lau=0
CREATE SQL VIEW lauto_view CONNECTION remote_01 SHARE ;
AS SELECT automobile_passenger_car.key_model, automobile_passenger_car.date_issue, ;
automobile_passenger_car.cost, account.selled ;
FROM automobile_passenger_car, account ;
WHERE automobile_passenger_car.key_auto=account.key_auto
ENDIF
IF smn=0
CREATE SQL VIEW sman_view CONNECTION remote_01 SHARE ;
AS SELECT salesman.key_salman, ;
salesman.last_name+' '+salesman.first_name+' '+salesman.patronymic as sman ;
FROM salesman
ENDIF
SELECT * FROM sman_view INTO ARRAY Arcombo1
IF cus=0
CREATE SQL VIEW cust_view CONNECTION remote_01 SHARE ;
AS SELECT customer.key_customer, customer.name_customer ;
FROM customer
ENDIF
SELECT * FROM cust_view INTO ARRAY Arcombo2
IF mod=0
CREATE SQL VIEW model_view CONNECTION remote_01 SHARE ;
AS SELECT model.key_model, model.name_model ;
FROM model
ENDIF
SELECT * FROM model_view INTO ARRAY Arcombo3
IF ord=0
CREATE SQL VIEW order_view CONNECTION remote_01 SHARE ;
AS SELECT order_.key_order, order_.key_salman, order_.key_customer, order_.key_model ;
FROM order_
&& Устанавливаем таблицу order_ обновляемой
=DBSETPROP('order_view', 'View', 'Tables', 'order_')
&& Устанавливаем имена для обновления
=DBSETPROP('order_view.key_order', 'Field', 'UpdateName', 'order_.key_order')
=DBSETPROP('order_view.key_salman', 'Field', 'UpdateName', 'order_.key_salman')
=DBSETPROP('order_view.key_customer', 'Field', 'UpdateName', 'order_.key_customer')
=DBSETPROP('order_view.key_model', 'Field', 'UpdateName', 'order_.key_model')
&& Задаем простой уникальный ключ на основе одного поля таблицы Order_
=DBSETPROP('order_view.key_order', 'Field', 'KeyField', .T.)
&& Задаем обновляемые поля
=DBSETPROP('order_view.key_salman', 'Field', 'Updatable', .T.)
=DBSETPROP('order_view.key_customer', 'Field', 'Updatable', .T.)
=DBSETPROP('order_view.key_model', 'Field', 'Updatable', .T.)
&& Активизация процесса обновления
=DBSETPROP('order_view', 'View', 'SendUpdates', .T.)
&& Задаем сравнение временной метки всех полей записи,

```

```
&& расположенной на удаленном источнике данных
=DBSETPROP('order_view', 'View', 'WhereType', 4)
ENDIF
```

```
USE order_view
```

```
=CURSORSETPROP("Buffering", 5)
```

Код для события Unload формы:

```
CLOSE DATABASES ALL
```

```
CLOSE TABLES ALL
```

Добавим в форму метод Combo_init и запишем для него следующий код:

```
nnAddItem="ThisForm.Combo"+ALLT(cNum)+".AddItem"
```

```
nnArn="Arcombo"+ALLT(cNum)
```

```
FOR I=1 TO ALEN(&nnArn,1)
```

```
&nnAddItem(allt(&nnArn(i,2)))
```

```
ENDFOR
```

Добавим в форму метод Refresh_list и запишем для него следующий код:

```
ThisForm.Combo1.Value=((ASCAN(Arcombo1,order_view.key_salman))+1)/2
```

```
ThisForm.Combo2.Value=((ASCAN(Arcombo2,order_view.key_customer))+1)/2
```

```
ThisForm.Combo3.Value=((ASCAN(Arcombo3,order_view.key_model))+1)/2
```

Добавим в форму метод Show_sav_und и запишем для него следующий код:

```
MyUpdate=1
```

```
ThisForm.mygo1.Com8.Enabled=.T.
```

```
ThisForm.mygo1.Com9.Enabled=.T.
```

Для события Click кнопки Propmodel (характеристика модели автомобиля) запишем следующий код:

```
glkey_mod=order_view.key_model
```

```
DO FORM prop_mod.scx
```

Для события Click кнопки Lauto (список автомобилей) запишем следующий код:

```
glkey_mod=order_view.key_model
```

```
glname_mod=ALLT(Arcombo3(ThisForm.Combo3.Value,2))
```

```
DO FORM listauto.scx
```

Для события Init всех элементов управления типа Combo Box (Combo1, Combo2 и Combo3) запишем следующий код:

```
cNum=RIGHT(This.Name,1)
```

```
ThisForm.combo_init
```

Для события InteractiveChange элемента управления Combo1 запишем следующий код:

```
REPLACE order_view.key_salman WITH Arcombo1(ThisForm.Combo1.Value,1)
```

```
ThisForm.Text2.Refresh
```

```
ThisForm.Show_sav_und
```

Для события InteractiveChange элемента управления Combo2 запишем следующий код:

```
REPLACE order_view.key_customer WITH Arcombo2(ThisForm.Combo2.Value,1)
```

```
ThisForm.Text3.Refresh
```

```
ThisForm.Show_sav_und
```

Для события InteractiveChange элемента управления Combo3 запишем следующий код:

```
REPLACE order_view.key_model WITH Arcombo3(ThisForm.Combo3.Value,1)
```

```
ThisForm.Text4.Refresh
```

```
ThisForm.Show_sav_und
```

Как видно из кода события Click кнопки lauto, сначала мы определяем значения кода (glkey_mod) и наименования (glname_mod) выбранной модели для дальнейшей выборки списка автомобилей по данной модели. После чего запускаем форму LISTAUTO (Список автомобилей). Похожая ситуация и с событием Click кнопки propmodel, где мы определяем значение кода (glkey_mod) модели и запускаем форму PROP_MOD (Характеристика модели автомобиля).

Кратко опишем вызываемые формы.

Форма LISTAUTO (Список автомобилей).

Некоторые формы используют данные из одной или нескольких таблиц. Для включения требуемых таблиц в описание формы выберите команду *Data Environment* из меню *View* главного меню Visual FoxPro. Откроется окно Data Environment, напоминающее область просмотра таблиц Конструктора базы данных.

Для добавления таблицы следует выбрать команду *Add* в меню *Data Environment*, которое добавляется в главное меню Visual FoxPro, или в контекстном меню, появляющемся при нажатии правой кнопки мыши. После этого выберите требуемую таблицу из диалогового окна *Add Table or View*.

Обратите внимание, что можно либо использовать несколько таблиц и определять отношения между ними в самой форме, либо использовать готовые многотабличные представления данных.

Для установления отношения между двумя таблицами нажмите на имя поля в главной таблице

и перенесите его в подчиненную таблицу. При этом между таблицами появится линия, как показано на рис. 9.10. Заметим, однако, что для этого необходимо заранее определить соответствующие индексы.

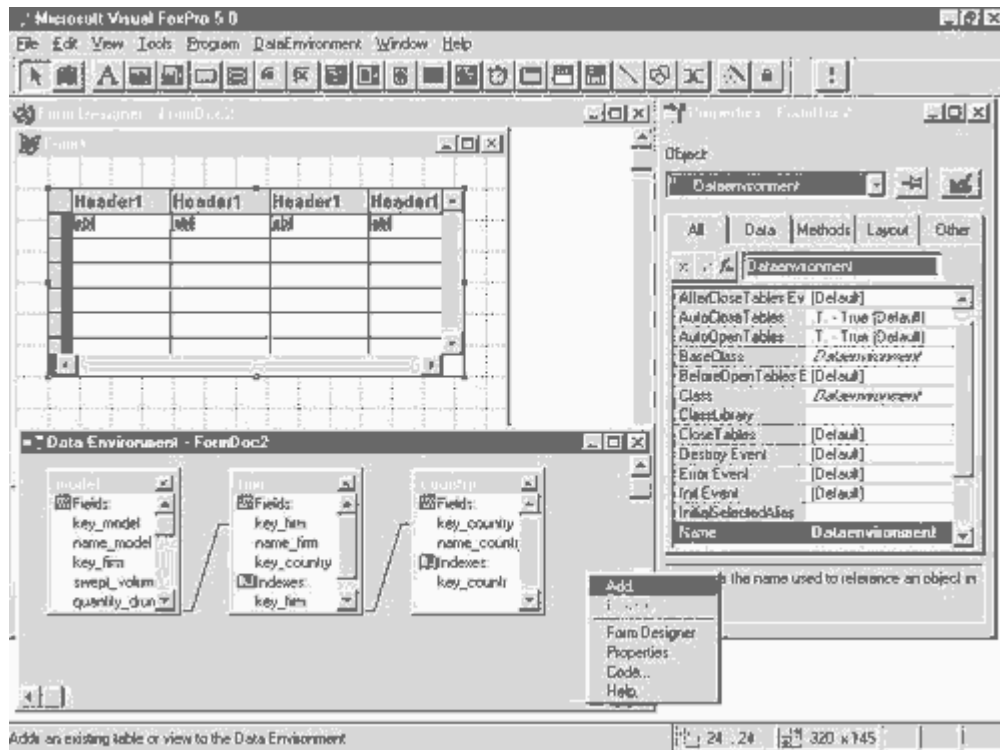


Рис. 9.10.

В форме LISTAUTO мы используем данные из представления lauto_view, и одним из возможных вариантов решения задачи является включение представления lauto_view в окружение формы (Data Environment) вышеизложенным способом. И этот вариант был бы не худшим. Однако мы в своем примере поступаем по-другому. А именно, включаем в событие, имеющее место непосредственно перед созданием формы (Form.Load), запрос к представлению lauto_view с выборкой данных по текущей модели, поместив результат запроса в курсор lauto. Подобный вариант приемлем только для просмотра данных. Дело в том, что данные в курсоре редактированию не подлежат.

В событие Load формы запишем следующий код:

```
SELECT lauto_view.date_issue, lauto_view.cost, ; lauto_view.selled ;
FROM lauto_view ;
WHERE lauto_view.key_model=glkey_mod ;
INTO CURSOR lauto
```

Далее, посредством панели инструментов Form Controls, в форму LISTAUTO включаем объект - таблицу (Grid).

В свойство источника данных (RecordSource), к которому привязан элемент управления Grid, помещаем имя курсора - lauto. После чего форму уже можно запускать на выполнение. И результат будет положительным. Хотя мы думаем, наш читатель вряд ли из тех, кого удовлетворит подобный результат. Дело в том, что заголовки колонок примут значения наименований полей курсора, элемент управления для отображения значений во всех колонках будет текстовым, что не всегда удобно, и т. д.

Поэтому мы несколько усовершенствуем данную форму.

На вкладке Layout для элемента управления Grid задаем количество объектов Column (свойство ColumnCount = 3), ликвидируем отображение столбца маркеров удаления (свойство DeleteMark = .F.), задаем тип полос прокрутки (свойство ScrollBars = 2 - только вертикальная полоса), задаем высоту заголовков столбцов (свойство HeaderHeight = 25) и высоту строк в элементе управления Grid (свойство RowHeight = 25).

На вкладке Data для каждой колонки таблицы Grid.Column задаем источник данных, к которому привязывается объект:

```
Grid.Column1.ControlSours=lauto.date_issue
```

```
Grid.Column2.ControlSours=lauto.cost
Grid.Column3.ControlSours=lauto.selled
```

Для третьей колонки указываем элемент управления в объекте Column - CurrentControl = CheckBox, который будет использоваться для отображения значений активной ячейки. Предварительно поместив его посредством перетаскивания элемента CheckBox из панели управления Form Control в колонку таблицы. Для этой же колонки позволяем свойству CurrentControl распространяться на все ячейки объекта Column (свойство Sparse = .F.).

На вкладке Layout для каждого заголовка колонки таблицы Grid.Column.Header, задаем текст (свойство Grid.Column1.Caption = Дата выпуска, Grid.Column2.Caption = Стоимость, Grid.Column3.Caption = Продажа), а вид выравнивания текста задаем по центру (свойство Alignment = 2).

После чего наша форма LISTAUTO будет иметь вид, показанный на рис. 9.11.

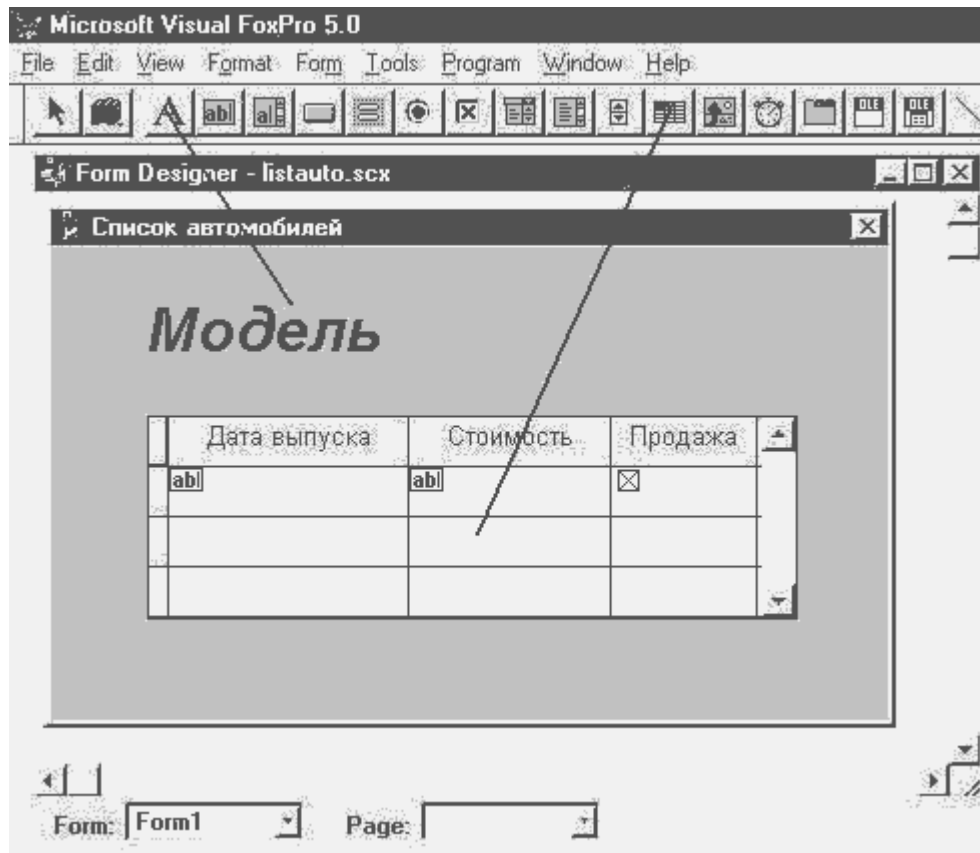


Рис. 9.11. Форма "Список автомобилей" в Конструкторе формы

Форма PROP_MOD (Характеристика модели автомобиля).

В форме PROP_MOD мы используем данные из представления propmodel_view, предварительно включив в событие, имеющее место непосредственно перед созданием формы (Form.Load), запрос к данному представлению с выборкой данных по текущей модели и поместив результат запроса в курсор propmodel.

В событие Load Формы запишем следующий код:

```
SELECT * FROM propmodel_view ;
WHERE propmodel_view.key_model=glkey_mod ;
INTO CURSOR propmodel
```

Далее с помощью панели инструментов Form Controls в форму PROP_MOD последовательно включаем объекты TextBox для всех полей курсора propmodel. Для каждого из них задаем источник данных (свойство ControlSours = <<имя_курсора.имя_поля>>). На рис. 9.12. показана форма PROP_MOD.

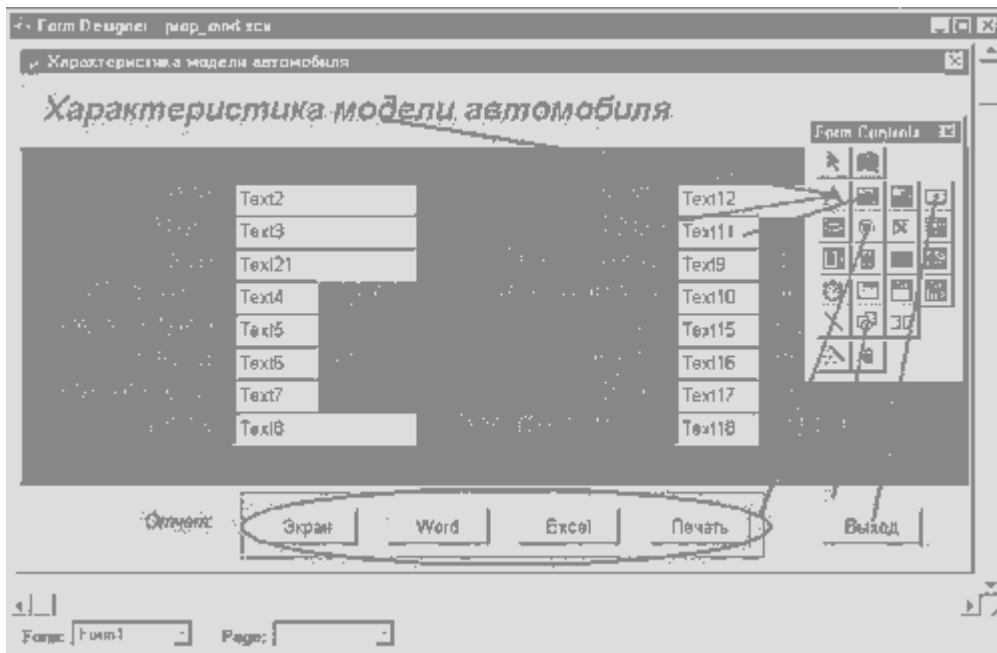


Рис. 9.12. Форма "Характеристика модели автомобиля" в Конструкторе формы

На примере данной формы нам бы хотелось показать, каким образом можно определять порядок перехода по объектам. Дело в том, что каждый объект имеет свой порядковый номер, по умолчанию совпадающий с порядком добавления объектов при создании формы. И при переходе с одного объекта на другой с помощью клавиатуры, порядок следования соответствует порядковым номерам. Однако это не всегда совпадает с логикой ввода данных.

Изменять порядок следования полей можно двумя способами: интерактивно (Interactive) и с помощью списка (By List). Требуемый метод выбирается во вкладке Forms в раскрывающемся списке Tab Ordering диалогового окна Options (см. рис. 9.5).

Если установлен метод интерактивного изменения порядка активизации полей в форме, то при выборе команды Tab Order из меню View главного меню Visual FoxPro в верхнем левом углу каждого объекта появится его порядковый номер (рис. 9.13).

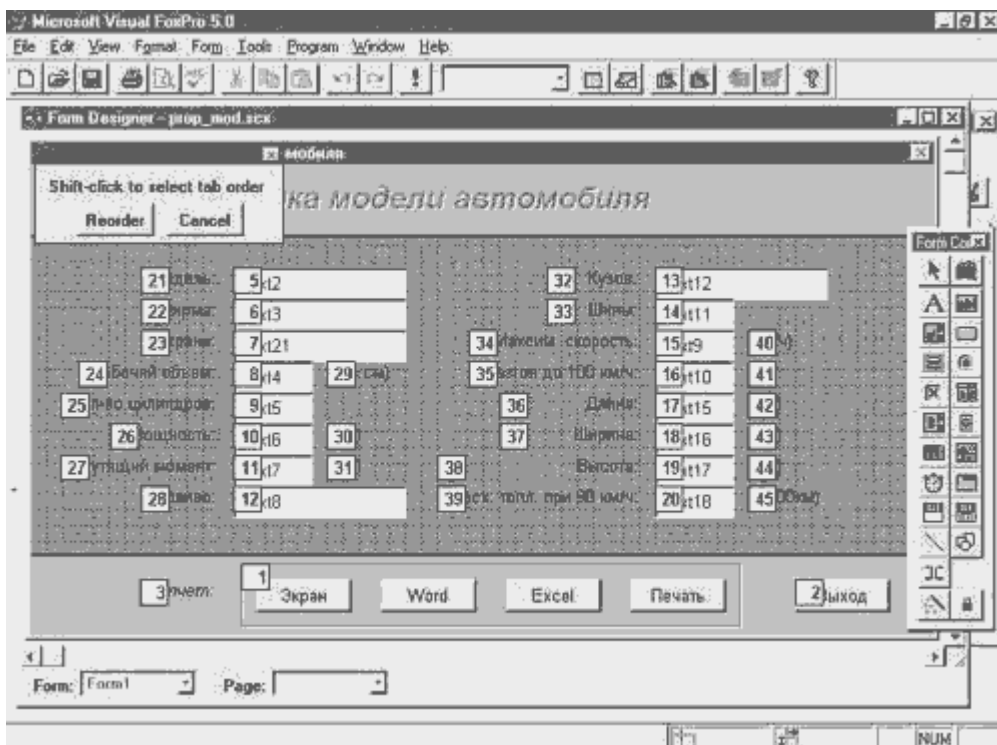


Рис. 9.13. Интерактивный метод изменения порядка активизации объектов в форме

Для изменения порядка перехода нажмите клавишу **Shift** и кнопку мыши над нужным объектом. При этом номер объекта будет убран, а все остальные объекты перенумерованы. При повторном нажатии объект будет помещен в конец списка. Очевидно, что применение данного метода для большой формы займет довольно много времени.

Более быстрый метод состоит в нажатии на первый объект в последовательности без нажатия на клавишу **Shift**. При этом объекту присваивается номер один, а все остальные номера исчезают. Затем, нажимая **Shift**, обойдите все остальные объекты в нужном порядке. Когда все сделано, нажмите кнопку **Reorder**.

Второй способ изменения порядка активизации полей - с помощью списка. В этом случае после выбора пункта меню **Tab Order** откроется диалоговое окно, показанное на рис. 9.14.

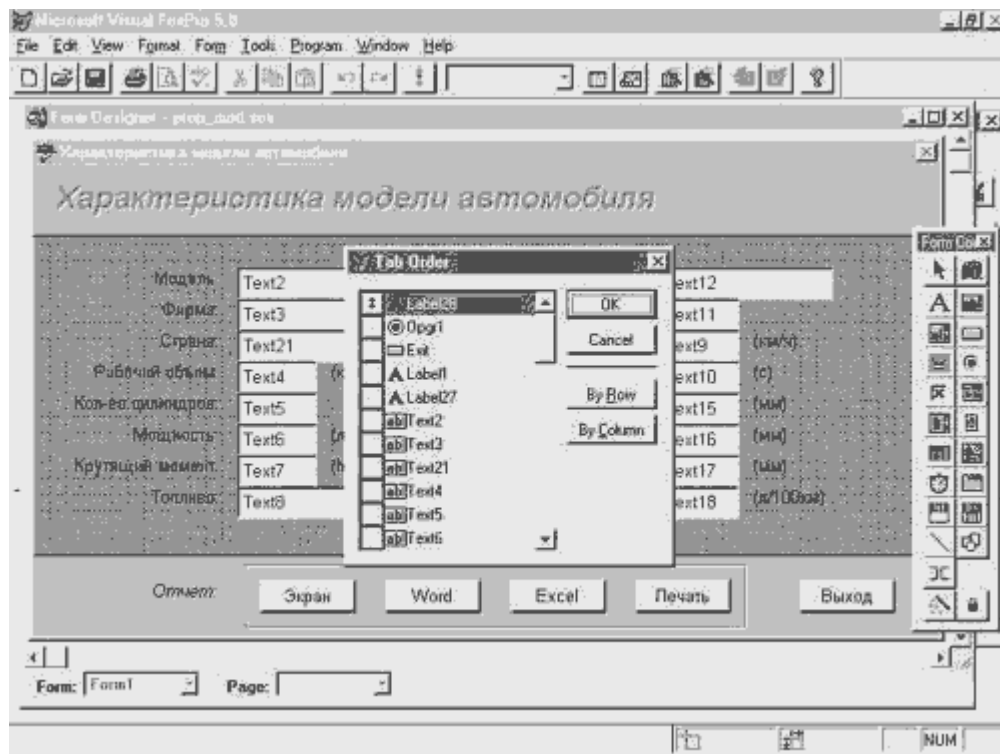


Рис. 9.14.

На левой стороне окна появляется прокручиваемый список всех объектов формы. Нажимая и перетаскивая кнопки слева от имен, можно как угодно изменять порядок активизации объектов.

Как видно из рис. 9.14, напротив каждого объекта находится маленькая пиктограмма, служащая для облегчения определения типа объекта. Кроме того, кнопки **By Row** и **By Column** позволяют быстро автоматически определить требуемый порядок, хотя на сложных формах их применение может и не дать желаемого результата. Добившись нужного порядка активизации объектов, нажмите кнопку **OK**.

В этой форме мы допустили возможность транспортировки данных в Microsoft Word и Microsoft Excel (с построением диаграммы). Как мы это сделали, будет рассказано в [следующей главе](#).

Описанные в данной главе формы используются в примере **SAMPLE.EXE**, поставляемом на диске.

Создание формы "Прием заказов" на Access

Мы уже неоднократно упоминали о широких возможностях Access как хорошего помощника в нелегком труде программиста. Здесь мы сосредоточимся на создании относительно сложной формы, при проектировании которой только Мастерами не обойтись.

Так же как в Visual FoxPro, основным инструментом для проектирования формы в Access является Конструктор формы, показанный на рис. 9.15. Для создания новой формы с помощью Конструктора формы в контейнере БД активизируйте вкладку **Форма** и нажмите кнопку **Создать**. В появившемся диалоговом окне **Новая форма** выберите в списке пункт **Конструктор**.



Рис. 9.15. Конструктор формы в Access 7.0

При загрузке Конструктора формы в Access, как и в остальных средствах, поддерживающих визуальное программирование, автоматически загружаются панели инструментов, служащие для создания объектов в форме и установки свойств объектов.

Как правило, это панели Конструктор Форм, Панель форматирования и Панель элементов.

Для нашего примера мы объявили переменные объектного типа на уровне модуля для того, чтобы они были видны из всех процедур всех модулей текущей базы данных, а при создании ссылки на эту базу данных - и из других баз данных.

```
'Создание переменных рабочего пространства, которые будут использоваться
'для начала и конца транзакций
Public mywksp As Workspace
Public mywksp1 As Workspace
'Создание переменных наборов данных
Public rstOrd As Recordset
Public rstcust As Recordset
Public rstAc As Recordset
```

Объекты типа `Workspace` необходимы нам для начала, завершения и отката транзакций. Объекты типа `Recordset` будут использоваться для работы с необходимыми нам данными, так как в некоторых случаях нам придется использовать формы, которые не имеют источника данных и соответственно не могут иметь связанных с какими-либо полями элементов управления. Например, форма "Прием заказов" не имеет источника данных.

Прием заказа начинается с нажатия на кнопку с соответствующим названием - "Прием заказа". После этого выполняется следующий код:

```
Private Sub Кнопка1_Click()
'Это строка условия поиска продавца в таблице Salesman
Dim strSearch As String
'Инициализируем переменную рабочего пространства
Set mywksp = DBEngine.Workspaces(0)
'Инициализируем переменную набора данных
Set rstOrd = mywksp.Databases(0).OpenRecordset("order_", _ dbOpenDynaset)
'начинаем транзакцию
mywksp.BeginTrans
'Строка для поиска кода продавца создается из названия поля
' Last_name и login текущего пользователя приложения
strSearch = "Last_name=" & Chr(34) & _ mywksp.UserName & Chr(34)
'Добавляем запись
rstOrd.AddNew
```

```

'Идентификатор продавца заносится в новую добавленную запись
rstOrd!key_salman = DLookup("key_salman",_"salesman", strSearch)
rstOrd.UPDATE
Me!cmbCust.Enabled = True
Me!txtFam.Enabled = True
End Sub

```

В результате выполнения этого кода в таблицу order_ вносится одна запись, но, обратите внимание, началась транзакция. Эти изменения еще не окончательные, мы можем при определенных обстоятельствах сделать откат. При работе программы предполагается, что каждый продавец, оформляющий сделку, будет запускать приложение со своим пользовательским именем, а с помощью этого имени будет находиться идентификатор, который и попадет в новую запись таблицы order_. Для этого мы используем свойство Username объекта mywksp.

```

strSearch = "Last_name=" & Chr(34) & mywksp.UserName & _ Chr(34)
'Добавляем запись
rstOrd.AddNew
'Идентификатор продавца заносится в новую
'добавленную запись
rstOrd!key_salman = DLookup("key_salman",_"salesman", strSearch)

```

После чего необходимо заняться поисками покупателя в предположении, что он уже совершал покупку до этого, либо ввести данные о нем в базу данных. При этом данные о покупателе будут сохраняться даже в том случае, если покупка не будет совершена, так как в дальнейшем вы можете использовать его адрес и имя для рассылки рекламных проспектов, приглашений на презентацию, для статистической обработки (жители какого квартала действительно покупают автомобили, а какого любят просто заходить в магазин) и для прочих нужд (зайти к клиенту в гости, находясь по соседству). С этой целью мы начинаем параллельную транзакцию в коде события Click, кнопки с подписью "Новый клиент":

```

Private Sub cmbNcl_Click()
Set mywksp1 = DBEngine.Workspaces(0)
Set rstcust = mywksp1.Databases(0).OpenRecordset("customer", _ dbOpenDynaset)
mywksp1.BeginTrans
DoCmd.OpenForm "Клиенты", acNormal, , , acAdd
End Sub

```

Эта транзакция завершится уже в форме "Клиенты". При этом, несмотря на все прекрасные перспективы дальнейшего использования данных о клиенте, существует возможность отката и в этом случае, например, если клиент вдруг неожиданно скроется, не успев назвать свой адрес и факс. Без них он для вас неинтересен.

Если клиент настаивает на том, что он совершал у вас покупку, то можно поискать его в вашей базе данных с помощью комбинированного списка. Применение фильтра ускорит поиск, фильтр можно установить с помощью единственного текстового поля, которое присутствует в нашей форме. Этого мы добиваемся с помощью кода, который выполняется при наступлении события LostFocus (Потеря фокуса):

```

Private Sub txtFam_LostFocus()
If Len(Trim(txtFam)) > 0 Then
Me!cmbCust.RowSource = "SELECT DISTINCTROW _
customer.key_customer,customer.name_customer,_
customer.address, customer.tel FROM customer Where _ customer.name_customer
like " & Chr(34) & Trim(txtFam) & _
""* " & _ Chr(34) & "; "
Else
Me!cmbCust.RowSource = "SELECT DISTINCTROW _
customer.key_customer,customer.name_customer,
_ customer.address, customer.tel FROM customer ; "
End If
End Sub

```

В данном коде используется свойство комбинированного списка - динамически изменять содержимое списка значений, другими словами, возможность использовать свойство RowSource

для чтения и записи.

Выбрав клиента, необходимо выбрать и автомобиль для него. Для этого предназначена форма "Наличие автомобилей, дата выпуска, цена", которая вызывается с помощью кнопки "Выбор автомобиля". Используется метод `OpenForm` объекта `DoCmd`:

```
Private Sub cmbCAuto_Click()
    DoCmd.OpenForm _
    "Наличие автомобилей, дата выпуска, цена", acNormal
End Sub
```

Форма "Наличие автомобилей, дата выпуска, цена" имеет установленное свойство `RecordSource` (Источник данных). Источником данных для этой формы служит запрос с одноименным названием. Для поиска автомобилей предназначены два комбинированных списка: с помощью первого вы отфильтровываете данные по названию производителя, а с помощью второго - по моделям. После этого из ограниченного списка данных вы легко выбираете автомобиль для вашего клиента. При этом используется замечательное свойство форм использовать фильтр и динамически менять его. Это становится ясно из следующего фрагмента кода. Как уже неоднократно отмечалось, с отфильтрованными данными Access работает значительно быстрее, особенно если вы их не изменяете, а просто просматриваете.

```
Private Sub name_model_AfterUpdate()
    If Len(Trim(name_model)) >> 0 Then
        Me.FilterOn = False
        Me.Filter = "name_model=" & Chr(39) & _
        & Me!name_model & Chr(39) & _
        "and name_firm=" & Chr(39) & Me!name_firm & Chr(39)
        Me.FilterOn = True
    End If
End Sub
```

После этого вы можете просмотреть подробную информацию о конкретной модели. Для чего вызывается еще одна форма, которая называется "Подробно о модели", устанавливается она с отфильтрованными данными.

```
Private Sub Кнопка9_Click()
    DoCmd.OpenForm "Подробно о модели", _
    acNormal, , "[Name_model]=" & Chr(39) & _
    Me![name_model] & Chr(39), acReadOnly
End Sub
```

После того как вы выберете автомобиль, подходящий вашему клиенту, можете передать управление форме "Прием заказов". При этом начинается заполняться таблица `Account`. Но, как вы помните, все это происходит в рамках транзакции, поэтому изменения еще не окончательны.

```
Private Sub Кнопка8_Click()
    Dim FStr As String
    FStr = "name_customer=" & Chr(39) & _ Forms![Прием заказов].cmbCust & Chr(39)
    Set rstAc = mywksp.Databases(0).OpenRecordset("account", _ dbOpenDynaset)
    rstAc.AddNew
    rstAc!key_customer = DLookup("key_customer", _ "customer", FStr)
    rstAc!key_auto = Me!key_auto
    rstAc!date_write = Now()
    rstAc.UPDATE
    Forms![Прием заказов].cmbAc.Enabled = True
    DoCmd.Close acForm, "Наличие автомобилей, дата выпуска, цена"
End Sub
```

После возврата в форму "Прием заказов" вы принимаете окончательное решение о продаже автомобиля. Пользователь имеет последнюю возможность отказаться. После нажатия на кнопку "Оформление счета" транзакция завершается.

```
mywksp.CommitTrans
rstOrd.Close
rstAc.Close
```

```
mywksp.Close
Me![Кнопка1].Enabled = True
DoCmd.OpenTable "account"
DoCmd.GoToRecord acTable, "account", acLast
```

Обратите внимание на то, что объекты `rstOrd`, `rstAc` и `mywksp` закрываются с помощью имеющегося у них метода `Close`. Если их не закрывать, то при нескольких повторных использованиях формы "Прием заказов" появится сообщение, что Access больше не может открыть таблицу.

Если покупатель все-таки решил не покупать автомобиль, то используйте кнопку "Отказ от заказа". При этом произойдет откат транзакции, и ни в одну из таблиц новые данные внесены не будут. Правда, есть один момент: если вы используете поле типа счетчик, то внутри базы данных его номер прирастет, поэтому при следующей попытке добавить запись, его значение будет больше на значение приращения.

```
Private Sub cmbRback_Click()
    mywksp.Rollback
    rstOrd.Close
    rstAc.Close
    mywksp.Close
    Me![Кнопка1].Enabled = True
    Me!cmbCust.Enabled = False
    Me!txtFam.Enabled = False
    Me!cmbAc.Enabled = False
End Sub
```

9.3. Разработка управляющего меню

Меню в прикладной программе - это первое, что видит пользователь, решив запустить разработанное вами приложение. Естественно, стоит приложить усилия, чтобы первое впечатление стимулировало, а не отпугивало пользователя.

В этом параграфе мы рассмотрим вопросы, связанные с разработкой меню для управления работой приложения.

Основное назначение меню заключается в том, чтобы дать возможность пользователю получить легкий доступ ко всем элементам прикладной программы. Следовательно, меню должно отображать не функциональную схему вашей программы и сложные взаимосвязи между ее блоками, а соответствовать логике работы пользователя. При разработке меню, на наш взгляд, следует придерживаться следующих принципов:

- Заголовок меню должен включать максимально ясную информацию о его назначении. Старайтесь использовать общеупотребительные термины, не используйте легкозабываемые сокращения.
- Структура меню должна соответствовать частоте выполнения тех или иных действий, логической последовательности их выполнения или в крайнем случае хотя бы алфавитному порядку. В то же время, если ваше меню насчитывает более 8 команд, алфавитный порядок их расположения может оказаться наиболее эффективным.
- Старайтесь выделять функционально связанные группы команд с помощью разделителей.
- Избегайте чрезмерно длинных списков команд в меню, делите такие списки на подменю по функциональным признакам. Во всяком случае, категорически избегайте создавать в меню списки команд, не уместающиеся на экране.
- Для часто используемых команд в меню определяйте горячие клавиши. Это позволит пользователю выполнять такие команды, не отрывая рук от клавиатуры.

Разработка меню в Visual FoxPro

Для разработки меню в Visual FoxPro проще всего использовать **Конструктор меню** (Menu Designer). Интересно отметить, что с помощью Конструктора меню можно не только разрабатывать меню для пользовательского приложения, но и настраивать меню Visual FoxPro для наиболее эффективной работы программиста.

В проекте выберем вкладку `Other`, найдем заголовок `Menus` и дадим команду `New`. На экране появится окно Конструктора меню, внешний вид которого приведен на рис. 9.16. На этом же рисунке поясняются основные элементы этого Конструктора. Обратите внимание, что Конструктор меню принципиально отличается от других визуальных средств проектирования

Visual FoxPro. После того как мы опишем с его помощью меню, необходимо сгенерировать программу. Файл с этой программой будет иметь расширение MPR, а файл после компиляции - MPX. Этот файл и следует запускать для работы с созданным меню. Недаром раньше такой инструмент программиста так и назывался - Генератор. Для генерации программы в меню *Menu* необходимо выбрать команду *Generate*.

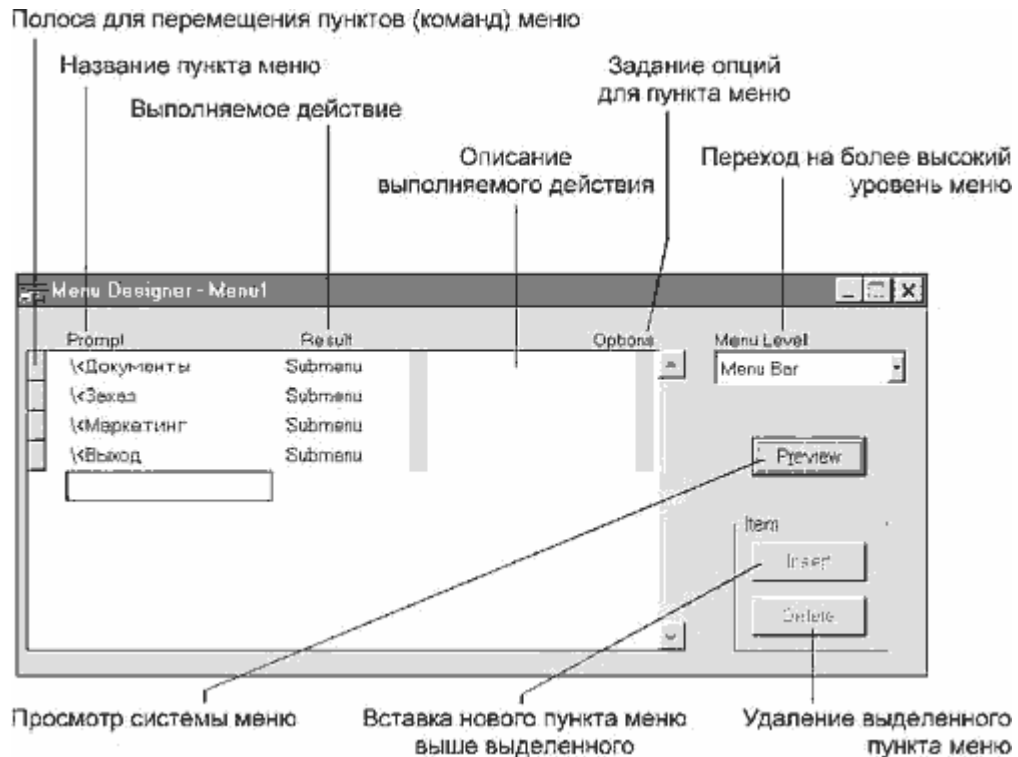


Рис. 9.16. Конструктор меню в Visual FoxPro

Если в вашу задачу входит настройка главного меню Visual FoxPro, то самое подходящее в этом случае решение - в меню *Menu* выбрать команду *Quick Menu*. После этого в Конструкторе меню в качестве заготовки мы получаем копию существующего меню Visual FoxPro, с которым можем проделывать любые изменения, включая удаление каких-либо меню и команд, добавление новых и т. д. Не стоит слишком усердствовать с удалением, помните, что, изымая какую-либо команду из меню, вы теряете предусмотренную наличием этой команды функциональность. Например, удалив меню *Edit*, вы не сможете использовать стандартные возможности переноса, копирования и поиска при работе с текстом. В соответствии со сложившимися правилами включать свои собственные меню в главное меню следует перед меню *Help*, которое должно оставаться последним.

Для разработки меню пользовательской программы нам придется вручную описать все необходимые пункты. Рассмотрим последовательность, в которой следует выполнить эти действия.

Запишем заголовки пользовательского меню в столбце *Prompt*. Если после выбора меню на экране должен появиться список команд, выберем в столбце *Result* пункт *Submenu* и щелкнем на появившейся справа кнопке *Create*. Мы окажемся на следующем уровне меню, где запишем заголовки команд, входящих в это подменю. Если структура нашего меню предусматривает еще один уровень вложения, повторите указанные действия. Для перехода с уровня подменю на верхний уровень необходимо выбрать его из комбинированного списка *Menu Level*. Если вы хотите выделить группы команд, поместите в столбце *Prompt* в строке, разделяющей группы команд меню, знаки \-. Вы всегда можете проверить, как визуально будет выглядеть ваше меню на экране, нажав клавишу *Preview*.

Для быстрого перемещения по меню можно назначить для его команд "горячие клавиши". Нажимая соответствующую алфавитную клавишу, пользователь может сразу выполнить нужную команду или перейти в какое-либо меню. Выбранная клавиша в меню подчеркивается. Для задания такой клавиши перед нужной буквой необходимо поставить знаки \<<. Например, если мы хотим, чтобы команда **Счет** выполнялась при нажатии клавиши **С**, мы должны заголовок этой команды написать в виде \<<**С**чет. После этого в меню буква С будет подчеркнута. Естественно, в одном меню не может быть несколько команд, использующих в качестве "горячих клавиш" одну и ту же букву. К сожалению, при использовании русских заголовков для меню не все так просто и, чтобы можно было реализовать описанные возможности, необходимо прочитать следующий абзац.

Наиболее часто выполняемым командам в меню можно присвоить клавиатурные комбинации. Для этого напротив соответствующего заголовка необходимо щелкнуть мышкой на кнопке в колонке **Options**. На экране появится диалоговое окно **Prompt Options**, с помощью которого для меню можно задать дополнительные условия. Внешний вид этого окна с необходимыми комментариями приведен на рис. 9.17. Щелкнем на поле проверки **Shortcut**, появится диалоговое окно **Key Definition**. Нажмем нужное сочетание клавиш. Их обозначение появится в поле **Key Label**. Символы в поле **Key Text** будут написаны рядом с командой меню, их можно отредактировать. Командам меню принято присваивать клавиатурные комбинации, начинающиеся с клавиши **Ctrl**. В клавиатурных комбинациях **Visual FoxPro** использует скан-код клавиши, поэтому не имеет значения текущий регистр и не важно, включена ли русская или латинская раскладка клавиатуры. Если вы используете знаки \<< для меню верхнего уровня, **Visual FoxPro** автоматически поместит в поле **Key Label** сочетание клавиш **Alt** и выделенной этими знаками буквы. Если буква русская, при запуске меню произойдет ошибка. Поэтому надо открыть окно **Key Definition** и вручную в поле **Key Label** исправить русскую букву на соответствующую той же клавише латинскую.



Рис. 9.17.

Вы можете регулировать доступ к тем или иным командам меню с помощью поля проверки **Skip For**, находящегося в диалоговом окне **Prompt Options**. Если вы щелкнете на этом поле, появится уже знакомое вам окно **Построителя выражения**, в котором можно сформировать условие доступа к этой команде меню. Если условие будет равно **.F.**, команда будет доступна, если **.T.** - недоступна. Наиболее часто эта возможность используется для регулирования доступа различных пользователей прикладной программы к тем или иным ее функциям.

В связи с тем, что **Visual FoxPro** поддерживает редактирование на месте **OLE**-объектов, с помощью поля проверки **Negotiate** мы можем для меню верхнего уровня указать, где меню будет располагаться после активизации **OLE**-сервера. По умолчанию действует установка **None** - меню убирается при редактировании **OLE**-объекта.

Для пояснения назначения той или иной команды меню с помощью поля проверки **Message** можно задать текст, который будет появляться в строке состояния. Для задания такого текста используется **Построитель выражения**. Если вы непосредственно набираете текст, не забудьте поместить его в кавычки.

В окне **Prompt Options** осталось еще одно поле проверки - **Pad Name**, о котором можно было бы ничего и не говорить, если бы мы не использовали русские заголовки для меню. При проектировании меню **Конструктор** автоматически присваивает имена меню на основании их заголовков. В то же время **Visual FoxPro**, как и его предшественники, на дух не переносит русских названий в меню. Не будем раздражаться по столь мелкому поводу, щелкнем на поле проверки **Pad Name** и изменим русское название на латинское.

Для выполнения каких-либо действий при выборе пользователем команд меню необходимо в колонке **Result** для каждой команды меню назначить команду **Visual FoxPro**, функцию или процедуру. Для назначения команды **Visual FoxPro** в столбце **Result** выберите пункт **Command** и наберите соответствующую команду в текстовом поле справа. Например, **DO Log_user**. Если указанная в этой команде процедура находится в блоке **Cleanup** **Конструктора** меню (о нем мы расскажем чуть позднее), то команду следует записать в виде: **DO Log_user IN Main**, где **Main** -

это имя файла меню.

Для задания при выборе команды меню выполнения какой-либо процедуры, в случае, когда меню не имеет подменю, необходимо в колонке **Result** выбрать пункт **Procedure**. Щелкнуть на кнопке **Create** и в появившемся окне поместить необходимый код.

В ряде случаев с помощью команд меню могут выполняться какие-то сходные для всех команд этого меню действия, не требующие написания большого объема кода. В этом случае мы можем задать при выборе команды меню выполнение одной процедуры для меню, имеющего подменю. Для этого все команды меню должны в колонке **Result** иметь пункт **Bar#**, который не предусматривает привязки к команде какого-то действия. В списке **Menu Level** выберем соответствующий уровень меню. Из меню **View Visual FoxPro** выберем команду **Menu Options**, после чего на экране появится одноименное диалоговое окно. Нажмем кнопку **Edit** и наберем код в соответствии с приведенным ниже шаблоном:

```
<<Код, выполняемый при выборе любой из команд>>
DO CASE
  CASE BAR() = 1
    <<Код, выполняемый для первой команды в меню>>
  CASE BAR() = 2
    << Код, выполняемый для второй команды в меню>>
...
ENDCASE
```

Функция **BAR()** возвращает номер выбранной команды меню. Вы можете при необходимости присвоить свои номера командам меню, используя поле справа от колонки **Result**.

Общие установки для системы меню можно выполнить, выбрав из меню **View Visual FoxPro** команду **General Options**. После появления диалогового окна с таким же названием, приведенного на рис. 9.18, можно написать программный код, который будет выполняться перед расположением меню на экране, выбрав поле проверки **Setup**. Процедуры, которые вы используете для выполнения команд меню, и действия, которые необходимо выполнить после исчезновения меню с экрана, можно записать, выбрав поле проверки **Cleanup**. С помощью группы кнопок выбора **Location** можно задать условия расположения меню после его запуска:



Рис. 9.18.

- Кнопка **Replace** - заменяет существующую систему меню.
- Кнопка **Append** - добавляет данное меню к существующей системе.
- Кнопка **Before** - располагает меню перед указанным в появляющемся справа списке меню.
- Кнопка **After** - располагает меню после указанного в появляющемся справа списке меню.

После запуска пользовательского меню вернуться к главному меню Visual FoxPro можно с помощью команды **SET SYSMENU TO DEFAULT**.

Если ваша пользовательская программа будет работать в виде самостоятельного модуля (EXE-файл) и вы планируете использовать меню в качестве главной программы вашего приложения, разместите команду **READ EVENTS** в блоке процедур Cleanup, а для команды меню, обеспечивающей прекращение работы приложения, задайте команду **CLEAR EVENTS**.

Разработка меню в Access

Для создания меню в Access используются два способа. Первый способ - это использование надстройки Построитель меню. Второй - использование Конструктора макросов. Первый способ наглядней и проще, но в итоге вы получаете тот же набор макросов, записанных с определенными параметрами и аргументами. Любое построенное меню вы можете подключить к любой форме или отчету с помощью свойства формы (отчета) Строка меню. Таким образом, в вашем приложении при переключении с одной формы на другую на экране будет отображаться меню, необходимое для решения данной задачи.

Вызовите надстройку Построитель меню с помощью команды *Надстройка меню Сервис*. На экране появится диалоговое окно, в котором вам будет предложено отредактировать уже существующее, создать новое или даже удалить существующее меню, как это показано на рис. 9.19.

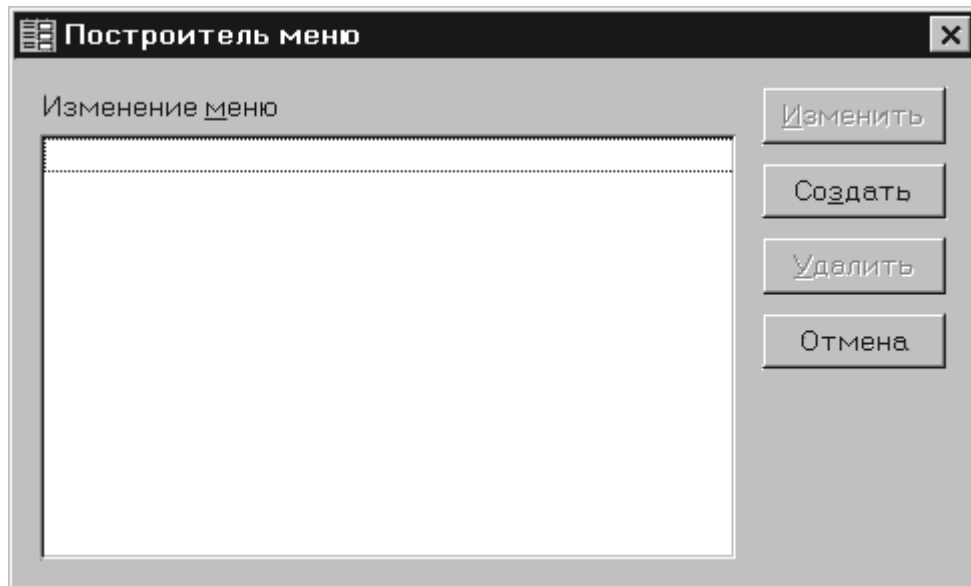


Рис. 9.19.

После выбора команды *Создать* путем нажатия кнопки с одноименным заголовком, возникает диалог, предлагающий выбрать шаблон для будущего меню. Если предполагается внести небольшие косметические изменения в стандартную линейку меню, то имеет смысл выбрать именно ее. Например, если при работе с формой вам нужно добавить один пункт в стандартную линейку меню, то, по-видимому, имеет смысл ее же и выбрать в качестве шаблона, как это показано на рис. 9.20.

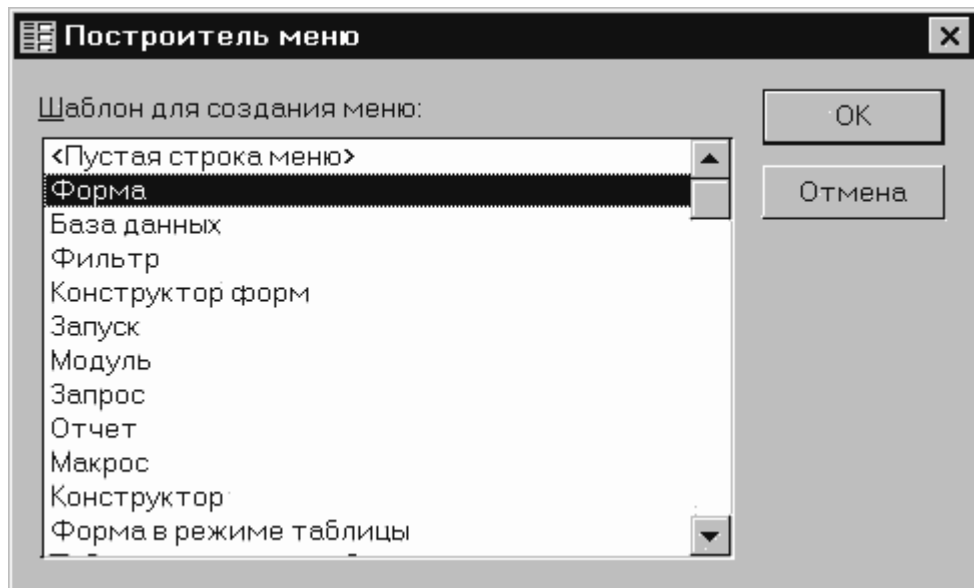


Рис. 9.20.

Следующим окном, если вы выбрали кнопку ОК в диалоге выбора шаблона, будет окно Конструктора меню. Это окно состоит из двух частей, как показано на рис. 9.21. В верхней части мы выбираем или вновь создаем название меню, команды меню или подменю. При работе с командой меню или подменю можно присвоить им макрокоманду, которая будет выполняться при ее выборе.

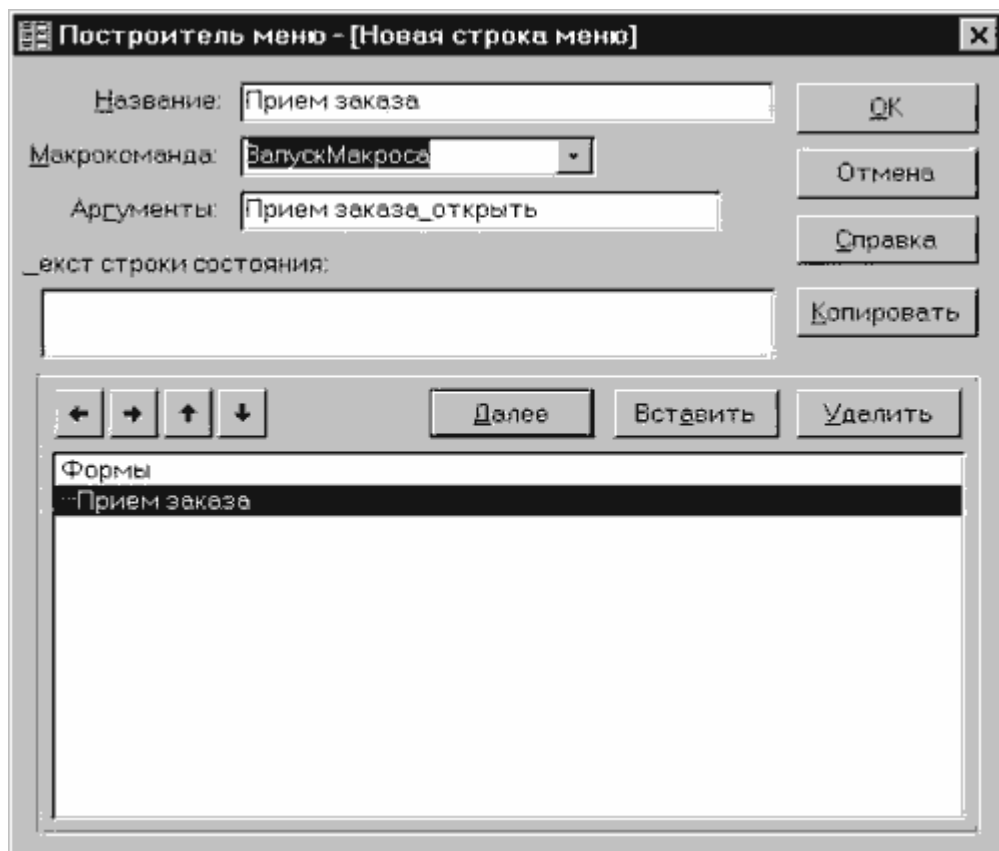


Рис. 9.21.

На первый взгляд выбор у нас не широк, так как этих макрокоманд всего три - "Запуск макроса", "Запуск программы" и "Команда меню". Но они позволяют запускать другие макрокоманды или процедуры, и поэтому круг возможных действий, которые может выполнить команда меню или подменю, значительно расширяется.

Чтобы запустить из нашего меню форму "Прием заказов", необходимо заранее подготовить

макрос или процедуру, которые будут выполнять эти действия.

В качестве примера создадим следующую очень простую процедуру:

```
Sub Прием заказа_открыть()
  DoCmd.OpenForm "Прием заказа"
End Sub
```

Вначале мы должны создать линейку меню, которая должна иметь по крайней мере один элемент. Пункты линейки меню отличаются от команд меню и подменю тем, что у них нет отступа в нижней части окна Конструктора меню.

На рис. 9.21 **"форма"** - это пункт линейки меню, а **"прием заказа"** - это команда меню, которая выполняет макрокоманду **"запустить программу"**, имеющую в качестве аргумента имя нашей процедуры **"Прием заказа_открыть"**.

Для того чтобы добавить или удалить отступ у элементов будущего меню, используйте кнопки со стрелками, которые находятся выше области отображения пунктов линейки, расположенной в нижней половине окна Конструктора меню.

Положение элемента в нижней половине окна диалога и наличие отступа определяют, будет ли он пунктом линейки меню, командой меню или подменю, а также месторасположение этого элемента в получившемся меню.

Названия команд меню должны следовать за названиями пунктов меню, к которым они относятся, аналогично названия команд подменю должны следовать вслед за командой меню, которая будет их выводить. Если вы хотите просто вывести разделитель в каком-либо из меню, то сделайте заголовком элемента дефис.

Сохранив Построитель меню, вы получите новые макросы, число которых будет равняться числу элементов меню за вычетом тех элементов, заголовками которых является дефис.

Глава 10

Использование готовых компонентов в приложении

10.1. Основные преимущества модульного проектирования прикладных программ

10.2. Как правильно использовать OLE 2.0

Возможности OLE 2.0

Использование OLE Automation

Управление объектами Excel

Управление объектами Word for Windows

10.3. Использование OLE Automation для передачи данных

Построение графиков с помощью MS Graph 5.0

Построение графиков с помощью MS Excel 7.0

Построение отчета в Word for Windows

Запись информации в Schedule+

10.4. Применяем ActiveX

Иерархический список

Календарь

Наверное, вы не раз с завистью смотрели на мощные графические средства популярных электронных таблиц или изощренные возможности редактирования текста в текстовом процессоре. Вы наверняка сможете написать такую же программу, - но стоит ли повторять уже сделанное? Гораздо привлекательнее научиться использовать готовые функциональные блоки в пользовательском приложении.

10.1. Основные преимущества модульного проектирования прикладных программ

В условиях жесткой конкуренции на рынке стандартного и заказного программного обеспечения большое значение имеет умение программиста предложить заказчику решение, которое позволит организации максимально быстро повысить эффективность и производительность ее работы.

В этом параграфе мы расскажем о том, какие возможности дает среда **Windows** для решения поставленной в предыдущем абзаце цели и постараемся убедить вас в эффективности модульного проектирования пользовательского приложения.

Одно из наиболее часто используемых решений - ориентировка на заказные прикладные программы, которые дают конечным пользователям наилучший доступ к информации с учетом всех особенностей работы данного заказчика. Однако создание заказных прикладных программ может быть очень дорогостоящим и наверняка потребует больших затрат времени. К тому же, в соответствии с развитием организации или какими-то изменениями в профиле ее действия, заказное программное обеспечение придется постоянно корректировать. Если посмотреть на заказную прикладную программу с точки зрения выполнения ею отдельных функций, то окажется, что значительная часть поддерживаемых ею функций, таких как редактирование данных, выполнение расчетов и т. п., достаточно стандартны и могут быть легко выполнены с помощью стандартного программного обеспечения. Использование функциональных возможностей, доступных в существующих прикладных программах, может существенно ускорить и удешевить процесс автоматизации обработки данных.

Например, прикладная программа, которая ищет с помощью системы управления базой данных сведения о потенциальных клиентах, может использовать возможности текстового процессора **Microsoft Word for Windows** для подготовки рекламных писем. Это решение позволяет использовать существующие функциональные возможности прикладных программ **Microsoft Office**, освобождая вас от необходимости разрабатывать всю прикладную программу.

Помимо несомненного выигрыша по времени, использование существующих прикладных программ в формировании заказных решений стимулируют следующие тенденции:

- Быстрое повышение мощности и снижение стоимости персональных компьютеров.
- Поистине революционные масштабы объединения вычислительных и информационных возможностей в рамках локальных и глобальных компьютерных сетей.
- Высокие требования пользователей к качеству пользовательского интерфейса. Для персонального компьютера графический пользовательский интерфейс стал в настоящее время стандартом де-факто.

Далее в этом и последующих параграфах данной главы мы рассмотрим наиболее важные аспекты, связанные с использованием функциональных возможностей **Microsoft Office** в прикладной программе для автоматизации обработки данных. Но сначала мы остановимся на тех средствах, которые составляют основу использования стандартного прикладного программного обеспечения в разрабатываемых пользовательских приложениях.

Стандартные прикладные программы **Microsoft Office** могут быть источником большого числа программных модулей (объектов) и составной частью инструментальных средств разработки благодаря функциональным возможностям стандарта **OLE 2.0** и языка программирования **Microsoft Visual Basic for Application**. Это означает, что прикладные программы **Microsoft Office** содержат компоненты, которые вы можете использовать при разработке пользовательского приложения. Вы и пользователи ваших программ получаете тем самым следующие существенные преимущества:

- Более эффективно используется стандартное программное обеспечение, которое, как правило, уже применяется пользователями.
- Сокращается процесс обучения пользователей, которые используют уже имеющиеся навыки работы со знакомыми программами.

Использование стандарта **OLE 2.0** является основой для разработки компонентного программного обеспечения, потому что этот стандарт обеспечивает средства для определения объектов и их совместного использования различными программами. Например, табличный процессор **Microsoft Excel** - одна из первых прикладных программ, которая предоставляет богатый набор объектов **OLE**. Библиотека объектных модулей **OLE Microsoft Excel** включает свыше 120 объектов и около 2700 связанных с ними методов и свойств. Эти объекты доступны через любую прикладную программу или язык, который поддерживает стандарт **OLE 2.0**, и их использование позволяет воспользоваться всеми функциональными возможностями **Microsoft Excel** без необходимости обеспечения этой функциональности в своей прикладной программе.

10.2. Как правильно использовать OLE 2.0

Стандарт **OLE - Object Linked and Embedding** (связывание и внедрение объектов) собственно своим названием определяет, что речь в нем идет о компоновке объектов и правилах их совместного использования для достижения интеграции среди прикладных программ.

Представляя изображения, диаграммы, таблицы, фрагменты речи, документы и другие функциональные единицы программы как объекты, пользователи могут с большей легкостью объединять и обрабатывать данные из разных прикладных программ.

В этом параграфе мы изучим:

- основные преимущества использования технологии OLE;
- методику создания и использования OLE-объектов;
- принципы работы OLE Automation.

Какие преимущества дает пользователю и разработчику эта технология? Вот основные из них:

- Редактирование на месте позволяет работать с документом другого приложения Windows, не покидая пользовательского приложения. Необходимые элементы управления, включая меню, появятся на месте текущих, а после завершения работы вернутся на свое место элементы управления и меню пользовательского приложения. Это позволяет избежать необходимости ручной загрузки и перехода в другое приложение.
- Перетаскивание данных между приложениями помогает реализовать наиболее естественный для пользователя метод сбора различных данных для использования в одном документе.
- Технологии встраивания и связывания объектов дают возможность выбрать наиболее эффективный способ объединения разнородных данных в одном документе. При этом связывание позволяет не увеличивать без необходимости объем документа и использовать наиболее свежую версию данных, хранящуюся в отдельном файле. Встраивание обеспечивает хранение всех необходимых данных в одном приложении.
- Поддержка вложенных объектов позволяет в одном документе держать несколько зависимых объектов без необходимости обращения к нескольким приложениям. Например, встроенная в форму электронная таблица Excel может содержать встроенную в таблицу диаграмму. Изменения в электронной таблице повлекут изменения в диаграмме.
- OLE Automation позволяет в приложении программным путем устанавливать свойства и задавать команды для объектов другого приложения. Это дает большие возможности для управления процессом подготовки текстовых документов высокого качества, формирования графиков или выполнения расчетов.

Возможности OLE 2.0

Стандарт OLE 2.0 описывает правила интеграции прикладных программ.

Он фактически расширяет пользовательский интерфейс, делая его более интуитивно понятным. Стандарт OLE 2.0 не ставит пользователя перед необходимостью набора команд или выбора опций, предоставляя возможность выполнять работу более естественным способом, управляя объектами в интерактивном режиме на экране компьютера. Пользователь, вместо того чтобы изучать методы работы с прикладной программой или операционной системой, может сосредоточиться на более эффективной работе с данными и документами.

Технология OLE 2.0 особенно полезна потому, что устанавливает стандартный метод взаимодействия между прикладными программами при использовании различных объектов (документов, подготовленных разными прикладными программами). Она предлагает мощные средства для создания документов, получающих данные из разнообразных источников информации. Такие документы называются **составными документами**. Объекты, которые они содержат, могут включать почти любой тип информации, в том числе текст, растровые изображения, иллюстрации и фрагменты речи.

В технологии OLE 2.0 выделяются два главных типа данных, связанных с объектом: данные представления и локальные данные.

Данные представления объекта - это информация, нужная для представления объекта на экране компьютера, в то время как **локальные данные объекта** - это вся информация, необходимая для прикладной программы, чтобы редактировать объект.

Используя OLE 2.0 пользователь может также связывать или включать объект в документ.

Связывание - это процесс, при котором в документ будут помещены только данные представления объекта и ссылка (или указатель на местонахождение) на локальные данные.

Локальные данные, связанные с объектом, существуют в некотором другом файле на диске. Всякий раз, когда объект модифицируется прикладной программой, этот файл открывается и прикладная программа использует хранящиеся в нем локальные данные. Пользователь работает со связанным объектом так, как будто он полностью содержится внутри документа.

Включение объекта физически помещает данные представления объекта и локальные данные внутри документа.

Вся информация, необходимая для редактирования объекта, содержится в документе. Любой объект, который содержит другие объекты (как связанные, так и вложенные), называется контейнером. Контейнерами чаще всего являются составные документы.

Включение объектов хотя и увеличивает объем документа, но позволяет перемещать объект наряду с документом на другой компьютер и тем самым редактировать такие объекты на различных рабочих местах. Связанные объекты не могут "путешествовать" с документами за пределы локальной файловой системы компьютера, но они более эффективны, чем вложенные объекты, потому что один образец локальных данных объекта может служить источником для большого числа различных документов.

Использование OLE Automation

Один из важнейших элементов стандарта OLE 2.0 - **OLE Automation** - определяет способ управления командами прикладной программы из другой прикладной программы.

Прикладные программы, поддерживающие **OLE Automation**, имеют соответствующие объекты, которые так и называются - объекты **OLE Automation**, посредством которых вы можете управлять работой всех остальных объектов прикладной программы, используя возможности **Visual Basic for Application**.

OLE Automation для управления объектами использует OLE-серверы.

OLE-сервер - это программа, которая может предоставить другим программам возможность использовать свои объекты.

Программы, которые могут управлять объектами OLE-серверов, называются **OLE-клиенты** или **OLE-контроллеры**.

Например, в версии 3.0 **Visual FoxPro** может выполнять функции только OLE-контроллера. Мы можем управлять объектами OLE-сервера, но не можем предоставить в чье-либо управление объекты **Visual FoxPro**.

Большинство OLE-серверов являются так называемыми **серверами Out-of-Process**. Они являются исполняемыми программами и могут взаимодействовать как с 16-bit, так и с 32-bit OLE-контроллерами. Расплатой за это является невысокая скорость обмена данными и значительные потребляемые ресурсы памяти. Другой тип OLE-сервера называется **In-Process** и представляет собой DLL-библиотеку, которая динамически подгружается и выгружается по необходимости. Хорошим примером такого сервера является процессор баз данных СУБД Access 7.0. Обмен данными с этим типом OLE-сервера происходит значительно быстрее, но работать он может только с OLE-контроллером такой же разрядности.

В табл. 10.1 приводится важная для разработчика информация о некоторых OLE-серверах Microsoft.

Таблица 10.1. OLE-серверы Microsoft

OLE-сервер	Где найти информацию об объектах
------------	----------------------------------

Серверы Out-of-Process

Microsoft Schedule+ 7.0	-
Microsoft Graph 5.0	VBA_GRP.HLP
Microsoft Word 7.0	WRDBASIC.HLP
Microsoft Excel 7.0	VBA_XL.HLP

Серверы In-Process

Data Access Object	DAO.HLP, DAOSDK.HLP
SQL Distributed Management Objects	SQLBOOKS.MVB

В качестве примера приведем несколько OLE-объектов для Microsoft Excel:

- **Application** - запускает программу Microsoft Excel;
- **Workbook** - рабочая книга, которая включает отдельные листы - один файл формата Microsoft Excel;
- **Chart** - график в рабочей книге;
- **Worksheet** - рабочий лист в книге;
- **Range** - одна ячейка или диапазон ячеек на листе.

В таких больших приложениях, как Microsoft Excel, мы сталкиваемся с очень большим количеством объектов. Причем каждый объект занимает свое строго определенное положение в иерархии объектов. Поэтому, обращаясь к объекту, мы должны, соответственно, перемещаться по этой иерархической структуре. Наверху иерархии находится объект **Application**. Какие-либо события или действия, связанные с этим объектом, будут иметь отношение в целом к прикладной программе. Например, для того чтобы закрыть приложение, мы должны написать такую команду:

Application.Quit

Объект **Application** включает в себя большое количество других объектов. Например, вы можете использовать следующую команду для ссылки на текущую рабочую книгу, открытую в Microsoft Excel:

Application.Workbooks

Заметьте, что название объекта **Workbooks** приводится во множественном числе, потому что это ссылка на коллекцию объектов, в данном случае рабочих книг.

Объект коллекции - это набор объектов, имеющих общие свойства, события и методы, что позволяет сослаться на них как на единый объект.

Объекты коллекции позволяют значительно проще выполнять многие распространенные действия с прикладной программой. Например, для выполнения действий с каждым объектом в коллекции можно использовать оператор цикла:

```
* Организуем ссылку на запущенное приложение
oExlApp = GETOBJECT("Excel.Application")
nWrk = 1
* Определяем, сколько открыто рабочих книг
nCount = oExlApp.Application.WorkBooks.Count
FOR nWrk = 1 TO nCount
    * Выведем наименование каждой рабочей книги
    ? oExlApp.Application.WorkBooks.Item(nWrk).FullName
NEXT
* Закроем приложение
oExlApp.Application.Quit
```

Чтобы запустить этот пример, загрузите Excel, откройте несколько файлов. В Visual FoxPro наберите приведенный в примере код в программном файле. В окне *Command* сделать это нельзя

из-за команды **FOR**, которая не поддерживается в интерактивном режиме командного окна. Вам также может показаться, что в этом фрагменте лишней является переменная `nCount`. Это не так! При использовании **OLE Automation** не встраивайте ее код внутрь команд **Visual FoxPro**. Запустите программный файл.

Как вы могли заметить из приведенного примера, для ссылки на одну рабочую книгу в коллекции мы применяли метод `Item`, в котором используется номер элемента коллекции. Для ссылки на первую рабочую книгу в коллекции укажите:

```
Application.Workbooks.Item(1)
```

Для того чтобы закрыть первую книгу в коллекции:

```
Application.Workbooks.Item(1).Close
```

В свою очередь каждая рабочая книга содержит коллекцию рабочих листов, каждый рабочий лист - коллекцию ячеек и т. д. В программе сослаться на ячейку **A1** можно таким образом:

```
Application.Workbooks.Item(1).Worksheets.Item("Sheet1").Cells.Item(1,1)
```

Как видно из приведенных примеров, мы должны разбираться в коллекциях для того, чтобы иметь возможность спускаться "вниз" по иерархии объектов. На рис. 10.1 приведена схема такого перемещения между объектами **Microsoft Excel**.

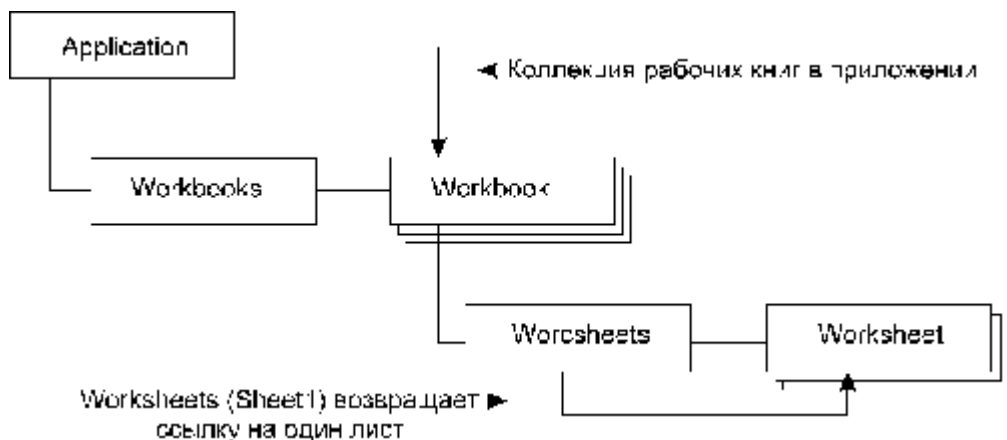


Рис. 10.1. Перемещение вниз по иерархии объектов с помощью коллекций в **Microsoft Excel**

Принцип перемещения вверх по иерархии более привычен для программиста, освоившего объектно-ориентированное программирование. Для этого используется свойство `Parent` и пока отсутствующее в **Visual FoxPro**, но привычное для приложений **Microsoft Office** свойство `Application`. Соответствующая схема для использования этих свойств приведена на рис. 10.2.

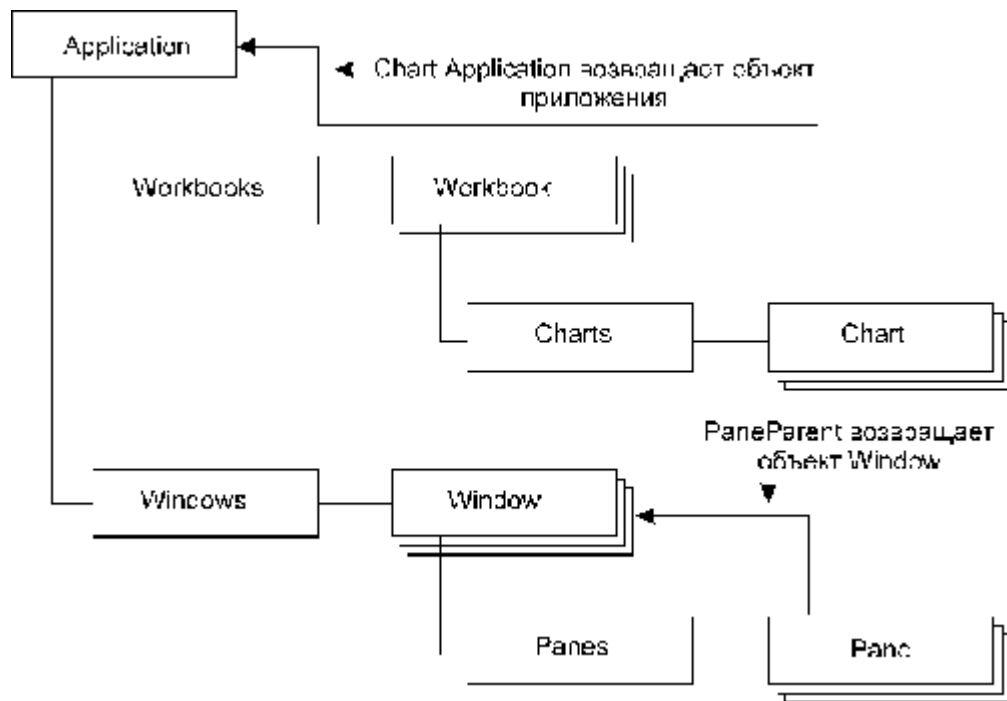


Рис. 10.2.

При использовании OLE Automation очень важным понятием является понятие объектов верхнего уровня.

В каждом приложении **объекты верхнего уровня** позволяют ссылаться на них из другой программы.

Имеющиеся, например, в Microsoft Excel объекты верхнего уровня представлены на рис. 10.3.



Рис. 10.3.

Несколько ранее вы, наверное, уже обратили внимание, что ссылка на вновь создаваемый объект OLE Automation в программе Visual FoxPro выполняется так же, как при не визуальном программировании на новый объект самого Visual FoxPro - с помощью функции

CREATEOBJECT(className [, eParameter1, eParameter2, ...])

Если OLE-объект уже существует, получить на него ссылку можно с помощью функции

GETOBJECT([FileName][, ClassName])

Параметр *FileName* позволяет указать имя существующего файла (и при необходимости путь к нему), содержащего OLE-объект, который необходимо активизировать. С помощью параметра *ClassName* можно указать имя объекта верхнего уровня (класс OLE-объекта). Это необходимо в том случае, если, как мы это показали на примере Microsoft Excel, в приложении в одном файле может храниться несколько объектов верхнего уровня, например таблицы, графики и т. д. В данном случае для параметра должен использоваться следующий синтаксис:

```
<<"Имя приложения">>.<<"Имя объекта верхнего уровня">>
```

Например:

```
oSht = GETOBJECT("C:\VFP\SAMPLE\VAT.XLS","EXCEL.SHEET")
```

В Visual Basic и соответственно Access мы можем использовать аналогичные функции, но для обеспечения ссылки на объект необходимо применять оператор **Set**, как показано в следующих примерах:

```
` В первом примере создаем объект MS Excel
Dim oExlApp As Object
' Объявляем переменную для ссылки на объект
Set oExlApp = CreateObject("Excel.Application")
` Создаем объект
oExlApp.Visible = True ' Выводим объект на экран
...
oExlApp.Quit ' После окончания работы закрываем Excel
Set oExlApp = Nothing ' Стираем ссылку на объект из памяти
` Во втором примере ссылаемся на объект MS Excel
Dim oExlApp As Object
Set oExlApp = GetObject("C:\VFP\SAMPLE\VAT.XLS","Excel.Sheet")
...
Set oExlApp = Nothing
```

Управление объектами Excel

Для того чтобы остановиться на проблемах взаимодействия пользовательского приложения с объектами Excel, нам придется более подробно обсудить специфические в данном случае особенности работы функций **CREATEOBJECT()** и **GETOBJECT()**. Поэтому систематизируем необходимые данные для важнейших объектов верхнего уровня Microsoft Excel и приведем их в табл. 10.2.

Таблица 10.2. Поведение объектов верхнего уровня Microsoft Excel

Объект	Функция	Описание поведения
Application	CreateObject	Всегда запускает невидимую копию Microsoft Excel. Файл рабочей книги не загружается.
	GetObject	Если параметр <i>FileName</i> представляет собой пустую строку, запускается новая невидимая копия Microsoft Excel без загрузки файла рабочей книги. Если параметр пропущен, то предпринимается попытка получить ссылку на уже запущенную копию Microsoft Excel и в случае неудачи генерируется ошибка. Не указывайте в параметре <i>FileName</i> имя файла. Для этого случая используйте объекты Sheet или Chart .
Sheet или Chart	CreateObject	Если не существует

Chart

запущенной копии Microsoft Excel, то запускается невидимая копия и создается рабочая книга с именем "Object" и один рабочий лист с именем "Sheet1". Для объекта Chart помимо этого создается еще один рабочий лист с именем "Chart1". Как только ссылка на созданный объект перестает существовать (стирается), рабочая книга удаляется, но Excel остается загруженным в памяти компьютера. Если одна или несколько копий Microsoft Excel уже запущены, то добавляется рабочая книга с параметрами, как это было описано в предыдущем абзаце. В случае, когда уже работает несколько копий Excel, предугадать, в какой из них будет добавлена таблица, невозможно.

GetObject

Если параметр *FileName* является допустимым (существующим) именем файла и Microsoft Excel не запущен, то запускается новая невидимая его копия с невидимой рабочей книгой. Если хотя бы одна копия Microsoft Excel уже запущена, рабочая книга открывается в ней. При этом будет получена ошибка, если указанное в функции имя файла совпадет с уже открытым файлом в загруженной копии Excel. Если параметр *FileName* задан в виде пустой строки (""), и если не существует запущенной копии Microsoft Excel, то запускается невидимая ее копия и создается рабочая книга с именем "Object" и один рабочий лист с именем "Sheet1". Если параметр *FileName* задан в виде пустой строки (""), и если хотя бы одна копия Microsoft Excel загружена в память, то в одной из них открывается рабочая книга, как это было описано выше. Если параметр *FileName* не указан, всегда генерируется ошибка.

Как вы можете заметить, изучив табл. 10.2, создание объектов OLE Automation или обращение к ним в Microsoft Excel зачастую может привести к совершенно различным последствиям. Почти

всегда Excel создает невидимую копию приложения. Для того чтобы сделать ее видимой для пользователя, надо присвоить свойству **Visible** объекта **Application** значение "истина":

```
...Application.Visible = .T.
```

Если создается невидимая копия рабочей книги, то сделать ее видимой можно, присвоив значение "истина" свойству **Visible** окна:

```
...Window(n).Visible = .T.
```

В некоторых случаях Excel при запуске посредством **OLE Automation** не открывает новый файл. Тогда это придется сделать с помощью метода **Add** коллекции **Workbooks**:

```
...Workbooks.Add
```

Для завершения обсуждения проблем, связанных с объектами верхнего уровня Microsoft Excel, приведем несколько примеров.

Объект Application

```
oApp = CREATEOBJECT("EXCEL.APPLICATION")
oApp = GETOBJECT(" ", "EXCEL.APPLICATION")
```

Любая из этих строк приведет к запуску новой копии Excel в скрытом виде.

```
oApp = GETOBJECT("EXCEL.APPLICATION")
```

Обеспечит ссылку на текущее приложение Excel. Если же Excel на компьютере не запущен, будет сгенерировано сообщение об ошибке.

Объект Sheet

```
oExSheet = CREATEOBJECT("EXCEL.SHEET")
FOR nVal = 1 TO 10
    oExlSheet.Cells(nVal, nVal).Value = nVal * nVal
ENDFOR
* Если хотите увидеть результат, уберите комментарий в следующий строке
* oExlSheet.Application.Visible = .T.
oExlSheet.SaveAs("TEMP.XLS")
```

Создает новый рабочий лист в запускаемой невидимой копии Excel и заполняет его цифрами так, как это показано на рис. 10.4. Созданная таблица сохраняется в файле **TEMP.XLS** в папке, установленной по умолчанию для Excel.

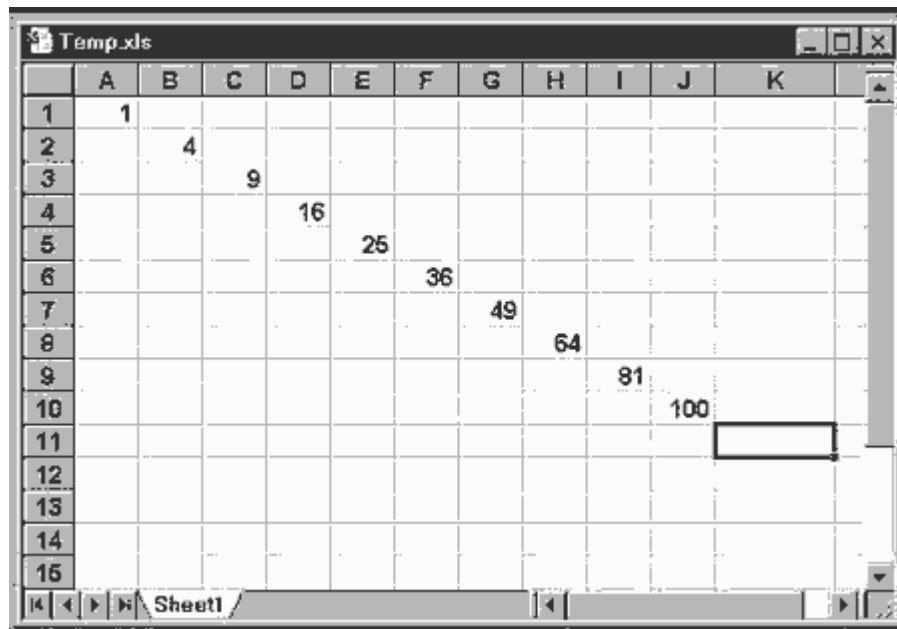


Рис. 10.4.

```
oExlSheet = GETOBJECT("RAS_1.XLS", "EXCEL.SHEET")
oExlSheet.Range("G2").Value = 16
oExlSheet.Range("G3").Value = 10
oExlSheet.Parent.Save && Сохраняем рабочую книгу
? oExlSheet.Range("G4").Value
```

Создает ссылку на первый рабочий лист в файле RAS_1.XLS. Файл в данном случае должен располагаться в папке, указанной по умолчанию для приложения, использующего OLE Automation, иначе необходимо указать путь к нему. В этот файл, содержащий результаты расчетов, мы заносим новые данные в ячейки G2 и G3, сохраняем данные, а затем выводим результат вычислений из ячейки G4, где записана формула, вычитающая G3 из G2.

Объект Chart

```
oExlChart = CREATEOBJECT("EXCEL.CHART")
* Стираем данные по умолчанию на рабочем листе со значениями
oExlChart.Parent.Sheets(2).Range("A1:D10").Clear
FOR nVal = 1 TO 10
    * Заполняем таблицу новыми значениями
    oExlChart.Parent.Sheets(2).Cells(nVal,1).Value = nVal
    oExlChart.Parent.Sheets(2).Cells(nVal,2).Value = nVal * nVal
NEXT
* Строим график по новым значениям по колонкам
oExlChart.ChartWizard(oExlChart.Parent.Sheets(2).Range("A1:B10"),,2)
* Смотрим, что получилось
oExlChart.Application.Visible = .T.
WAIT
```

Создается новый рабочий лист с диаграммой. Сначала мы заполняем необходимыми данными второй рабочий лист, на котором размещаются данные. Затем с помощью метода ChartWizard обновляем диаграмму на первом рабочем листе. В методе ChartWizard из большого числа параметров мы устанавливаем только область данных и условие построения его по колонкам. Остальные параметры принимают значения по умолчанию. После этого выводим Excel на экран и используем команду WAIT для того, чтобы рабочая книга не была выгружена после завершения работы программы.

```
oExlChart = GETOBJECT("RAS_1.XLS", "EXCEL.CHART")
oExlChart.ChartTitle.Text = "Результаты продаж"
```

Создает ссылку на первый рабочий лист с диаграммой в файле RAS_1.XLS. В файле должен быть предварительно создан объект Chart с помощью команды Chart в меню Insert.

Управление объектами Word for Windows

При работе с объектами Word for Windows функция **CREATEOBJECT()** запускает новую копию приложения только в том случае, если на компьютере это приложение еще не работает. При этом функция **GETOBJECT()** не может использоваться для ссылки на документ. Если она используется в виде

```
oWrd = GETOBJECT(" ", "WORD.BASIC")
    то ее применение аналогично функции CREATEOBJECT().
    Задание функции в виде
oWrd = GETOBJECT("WORD.BASIC")
```

всегда приводит к ошибке.

В отличие от Excel текстовый процессор Word for Windows имеет только один объект OLE Automation - **WordBasic** (в синтаксисе мы пишем Word.Basic). Это означает, что управлять документом Word из другого приложения мы можем только посредством выполнения команд WordBasic.

Для того чтобы продемонстрировать пример создания объекта Word, решим классическую для любого начинающего программиста задачу - выведем на экран слово "Hello!" В связи с тем, что читатель [одиннадцатой главы](#) этой внушительной по объему книги вряд ли захочет относить себя к начинающим программистам, усложним задачу. Загрузим на компьютере Word и Access. Теперь, находясь в Access, напомним заветное слово в документе Word.

Для решения этой задачи в контейнере БД Access перейдем на вкладку Модули. На панели инструментов нажмем кнопку Вставить процедуру. В появившемся диалоговом окне напомним ее название, например CallWord. В окне Модуль напомним следующий код:

```
Public Function CallWord() As Integer
    Dim oWbApp As Object
    Set oWbApp = CreateObject("Word.Basic")
    oWbApp.FileNew ` Открываем новый документ
    oWbApp.Insert "Hello!" ` Вписываем в документ слово
    Set oWbApp = Nothing
    Exit Function
End Function
```

Запустите этот модуль на выполнение. Напоминаем простейший способ сделать это - нажмем кнопку на панели инструментов Окно отладки и в появившемся окне наберите имя процедуры, после чего нажмем клавишу **Enter**. Смело переходите в Word и рассматривайте появившееся там слово.

Если вас испугало многообразие объектов в Excel и вы уже обрадовались наличию в Word всего одного объекта, то, возможно, ваша радость преждевременна. При управлении документом Word с помощью OLE Automation есть одна существенная для программиста неприятность.

Команды WordBasic используют поименованные аргументы, а посредством OLE Automation ссылаться на аргументы можно только по их положению. Например, вы хотите отключить вывод предупреждения при сохранении документа. В файле контекстной справки WordBasic вам будет подсказан следующий синтаксис команды ToolsOptionsSave

```
ToolsOptionsSave [.CreateBackup = number] [, .FastSaves = number] [, .SummaryPrompt = number]
```

В макросе Word вы, соответственно, можете написать:

```
ToolsOptionsSave .SummaryPrompt = False
```

Для того чтобы выполнить эту команду из OLE-контроллера, необходимо знать порядковый номер этого аргумента в команде:

```
oWbApp.ToolsOptionsSave , , 0
```

10.3. Использование OLE Automation для передачи данных

В практике создания систем автоматизации обработки данных программисту очень часто приходится сталкиваться с задачей графического представления данных. В комплекте со всеми средствами разработки Microsoft поставляется специальная утилита Microsoft Graph 5.0, которая как нельзя лучше подходит для этих целей.

В этом параграфе мы расскажем, как в пользовательском приложении можно использовать Microsoft Graph 5.0, а также остановимся на основных возможностях графического представления

данных, которые предоставляет Microsoft Excel.

С помощью Microsoft Graph 5.0 мы можем легко встроить в форму в виде OLE-объекта графики самого разнообразного вида, которые помогут пользователю лучше разобраться с тенденциями изменения интересующих его показателей. MS Graph 5.0 имеет интуитивно понятный интерфейс, разобраться с которым в крайнем случае поможет обширная справочная информация. Но если мы не хотим заставлять пользователя нашей прикладной программы изучать еще и англоязычный интерфейс MS Graph, нам придется позаботиться об управлении процессом графического представления данных из программы с помощью средств OLE Automation. В связи с актуальностью этого вопроса остановимся на нем возможно подробнее.

На рис. 10.5 представлена иерархическая схема объектов MS Graph 5.0, а в табл. 10.3 - их краткое описание.

Таблица 10.3. Описание объектов MS Graph 5.0

Объект	Описание
Application	Содержит установки в целом для приложения
Axis	Горизонтальные или вертикальные оси графика
AxisTitle	Заголовок одной из осей графика
Border	Рамка
Chart	График
ChartArea	Область графика, которая простирается и за внешние границы области, в которой вычерчивается график (поля, заголовки осей, описания данных и т. д.)
ChartGroup	Группа серий данных, представленных в виде одного графика. Простейший график содержит один объект ChartGroup
ChartTitle	Заголовок графика
DataLabel	Дополнительное описание для графика одной серии данных (может иметь отношение ко всей серии или к одной точке)
DownBars	Прямоугольник, который на линейном графике соединяет значение одной серии данных с более низким значением другой серии данных
DropLines	Вертикальные линии, соединяющие точки графика со значениями на оси Y
ErrorBars	Линии, графически отображающие ошибку расчета значения в графиках X-Y
Floor	Координатная поверхность, служащая основанием для трехмерного графика
Font	Шрифт
GridLines	Линии координатной сетки
HiLoLines	Вертикальные линии, которые соединяют максимальное и минимальное значения для различных серий данных
Interior	Графическое оформление (фон, тень и т. д.)
Legend	Описание серии данных для графика
LegendEntry	Наименование серии данных, которое берется из колонки с данными и может быть изменено только на листе с данными
LegendKey	Обозначения для серии данных
PlotArea	Область непосредственного расположения графика
Point	Точка в серии данных

Series	Серия данных
SeriesLines	Линия для обозначения серии данных
TickLabels	Значения для засечек на оси графика
TrendLine	Линия тренда
UpBars	Прямоугольник, который на линейном графике соединяет значение одной серии данных с более высоким значением другой серии данных
Walls	Боковые координатные поверхности для трехмерных графиков

Перечисленные 27 объектов позволяют достаточно гибко программировать внешний вид графиков, предоставляя доступные для них свойства и методы, список которых приведен в табл. 10.4. Все объекты имеют свойства **Application** (возвращает имя приложения, создавшего объект), **Creator** (возвращает идентификатор приложения, создавшего объект - для **Apple Macintosh**) и **Parent** (возвращает имя родительского объекта), поэтому они не указаны в таблице.

Таблица 10.4. Свойства и методы объектов MS Graph 5.0

Объект	Свойства	Методы
Application	ChartWizardDisplay, HasLink, Name, Visible, Chart, Quit	SaveAsOldExcelFile-Format
Axis	AxisBetweenCategories, AxisGroup, AxisTitle, Border, Crosses, CrossesAt, HasMajorGridlines, HasMinorGridlines, HasTitle, MajorGridlines, MajorTickMark, MajorUnit, MajorUnitIsAuto, MaximumScale, MaximumScaleIsAuto, MinimumScale, MinimumScaleIsAuto, MinorGridlines, MinorTickMark, MinorUnit, MinorUnitIsAuto, ReversePlotOrder, ScaleType, TickLabelPosition, TickLabels, TickLabelSpacing, TickMarkSpacing, Type	Delete
AxisTitle	Border, Caption, Font, HorizontalAlignment, Interior, Left, Name, Orientation, Shadow, Text, Top, VerticalAlignment	Delete
Border	Color, ColorIndex, LineStyle, Weight	-
Chart	Area3DGroup, AutoScaling, Bar3DGroup, ChartArea, ChartTitle, Column3DGroup, Corners, DepthPercent, DisplayBlanksAs, Elevation, Floor, GapDepth, HasAxis, HasLegend, HasTitle, HeightPercent, Legend, Line3DGroup, Perspective, Pie3DGroup, PlotArea, PlotVisibleOnly, RightAngleAxes, Rotation, SizeWithWindow, SubType, SurfaceGroup, Type, Walls, WallsAndGridlines2D,	BarGroups, ChartGroups, ChartWizard, ColumnGroups, Delete, DoughnutGroups, LineGroups, OmitBackground, PieGroups, RadarGroups, SeriesCollection, SetEchoOn, XYGroups

	ApplyDataLabels, AreaGroups, AutoFormat, Axes	
ChartArea	Border, Font, Height, Interior, Left, Name, Shadow, Top, Width	Clear, ClearContents, ClearFormats, Copy
Chart- Group	AxisGroup, DoughnutHoleSize, DownBars, DropLines, FirstSliceAngle, GapWidth, HasDropLines, HasHiLoLines, HasRadarAxisLabels, HasSeriesLines, HasUpDownBars, HiLoLines, Overlap, RadarAxisLabels, SeriesLines, SubType, Type, UpBars, VaryByCategories	SeriesCollection
ChartTitle	Border, Caption, Font, HorizontalAlignment, Interior, Left, Name, Orientation, Shadow, Text, Top, VerticalAlignment	Delete
DataLabel	AutoText, Border, Caption, Font, HorizontalAlignment, Interior, Left, Name, NumberFormat, Orientation, Shadow, ShowLegendKey, Text, Top, Type, VerticalAlignment	Delete
DownBars	Border, Interior, Name	Delete
DropLines	Border, Name	Delete
ErrorBars	Border, EndStyle, Name, Clear Formats	Delete
Floor	Border, Interior, Name	Clear Formats
Font	Background, Bold, Color, ColorIndex, FontStyle, Italic, Name, OutlineFont, Shadow, Size, Strikethrough, Subscript, Superscript, Underline	-
GridLines	Border, Name	Delete
HiLoLines	Border, Name	Delete
Interior	Color, ColorIndex, InvertIfNegative, Pattern, PatternColor, PatternColorIndex	-
Legend	Border, Font, Height, Interior, Left, Name, Position, Shadow, Top, Width	Clear, ClearContense, Copy, Delete, LegendEntries
Legend- Entry	Font, Index, LegendKey	Delete
LegendKey	Border, Interior, InvertIfNegative, MarkerBackgroundColor, MarkerBackgroundColorIndex, MarkerForegroundColor, MarkerForegroundColorIndex, MarkerStyle, Smooth	ClearFormats, Delete
PlotArea	Border, Font, Height, Interior, Left, Name, Top, Width	ClearFormats
Point	Border, DataLabel, Explosion, ApplyDataLabels,	

	HasDataLabel, Interior, InvertIfNegative, MarkerBackgroundColor, MarkerBackgroundColorIndex, MarkerForegroundColor, MarkerForegroundColorIndex, MarkerStyle, PictureType, PictureUnit	ClearFormats, Copy, Delete
Series	AxisGroup, Border, ErrorBars, Explosion, HasDataLabels, HasErrorBars	Interior, InvertIfNegative, MarkerBackgroundColor, MarkerBackgroundColorIndex, MarkerForegroundColor, MarkerForegroundColorIndex, MarkerStyle, PictureType, PictureUnit, Smooth, Type, ApplyDataLabels, ClearFormats, Copy, DataLabels, Delete, ErrorBar, Points, Trendlines
Series-Lines	Border, Name	Delete
TickLabels	Font, Name, NumberFormat, Orientation	Delete
TrendLine	Backward, Border, DataLabel, DisplayEquation, DisplayRSquared, Forward, Index, Intercept, InterceptIsAuto, Name, NameIsAuto, Order, Period, Type	Clear Formats, Delete
UpBars	Border, Interior, Name	Delete
Walls	Border, Interior, Name	ClearFormats

Построение графиков с помощью MS Graph 5.0

Возможности отображения данных с помощью MS Graph 5.0 рассмотрим на примере построения формы в Visual FoxPro, в которой пользователь мог бы анализировать изменение данных с помощью графиков различного типа. Внешний вид такой формы приведен на рис. 10.6. Форма включает внедренный объект MS Graph 5.0 для отображения графика, раскрывающийся список для изменения отображаемых данных, раскрывающийся список и две кнопки выбора для изменения типа графика, кнопку управления для закрытия формы и элементы внешнего оформления.

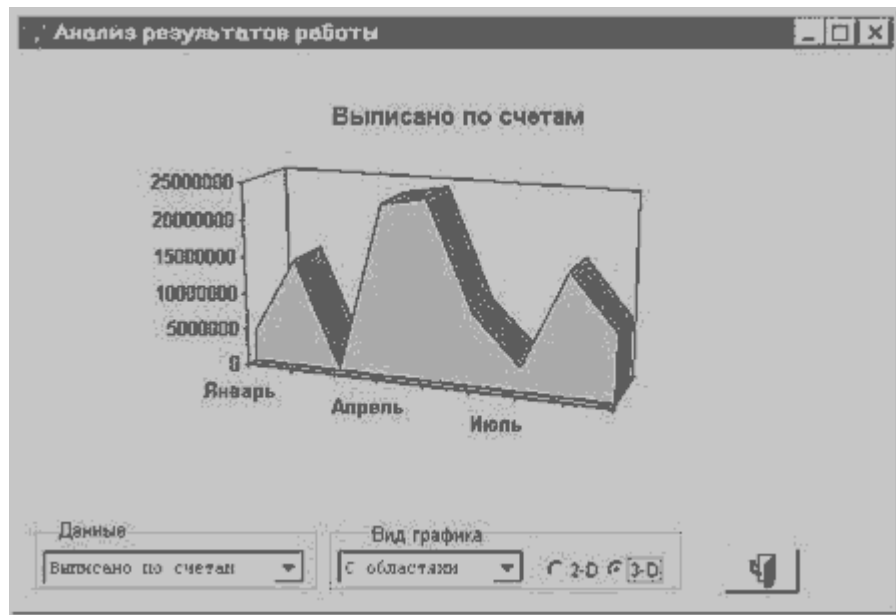


Рис. 10.6.

Для создания этой формы предварительно нам придется создать вспомогательную таблицу для хранения внедренного объекта MS Graph (графика). В этой же таблице (ее структура приведена в табл. 10.5) удобно хранить и дополнительную информацию о заголовках, а также сам запрос для получения наиболее "свежих" данных. Создать ее можно интерактивно с помощью Конструктора таблицы и информации, приведенной в табл. 10.5, или, что будет быстрее, с помощью следующих команд:

```
CREATE TABLE Graphs (title C(25), number I, query M, y_number I,;
  y1_caption C(25), y2_caption C(25), graph G)
APPEND BLANK
APPEND GENERAL graph CLASS "MSGraph"
```

Таблица 10.5. Структура таблицы GRAPH.S.DBF для хранения внедренного графика

Поле	Тип	Ширина	Назначение поля
TITLE	Character	25	Заголовок графика
NUMBER	Integer	4	Номер графика
QUERY	Memo	4	Команда SQL запроса
Y_NUMBER	Integer	4	Число осей Y
Y1_CAPTION	Character	25	Заголовок первой оси Y
Y2_CAPTION	Character	25	Заголовок второй оси Y
GRAPH	General	4	График в виде внедренного объекта MS Graph 5.0

В созданную таблицу запишите необходимые данные. Обратите внимание, что число символов в SQL запросе, записываемом в поле Query, не должно превышать 256 и он должен быть записан в одну строку. Для того чтобы уложиться в это число, не стоит пренебрегать никакими способами сокращения длины команды, в том числе полезно вспомнить о возможности использования внутренних псевдонимов, как это показано в следующем примере (здесь сознательно не используется символ переноса строк):

```
SELECT m.month_name,SUM(d.Price*d.quantity),m.month_num
FROM Months m,Invoices i,Inv_Details d WHERE i.kod_id = d.kod_id AND
MONTH(i.inv_date) = m.month_num GROUP BY m.month_name ORDER BY m.month_num
INTO CURSOR Datafile
```

Для временного хранения результата запроса мы используем курсор с именем Datafile. Теперь откроем Конструктор формы и разместим в новой форме объект OLE Bound Control.

Присвоим ему имя `olbGraph`. В `Data Environment` формы добавим только что созданную таблицу и для свойства `ControlSource` объекта `olbGraph` выберем поле `graph`.

В коде события `Init` обновим данные для отображаемого графика:

```
ThisForm.LockScreen = .T.
SELECT Graphs
GO TOP
cCom = Query
cCom = LEFT(cCom, LEN(cCom) - 2)
* Выполняем запрос, записанный в поле примечаний Query
&cCom
ThisForm.RefreshGraph
ThisForm.ChangeGraphType
ThisForm.LockScreen = .F.
```

В коде для события `Load` формы опишем массив `aType` возможных типов графиков. В дальнейшем этот массив будет источником данных для списка типов графиков:

```
PUBLIC ARRAY aType[4,2]
aType[1,1] = "С областями"
aType[2,1] = "Гистограмма"
aType[3,1] = "График"
aType[4,1] = "Круговая"
aType[1,2] = 1
aType[2,2] = 3
aType[3,2] = 4
aType[4,2] = 5
```

Тогда в событии `Destroy` формы не забудем стереть этот массив:

```
RELEASE aType
```

Для отображения названий графиков создадим раскрывающийся список с именем `cmbData`. Присвоим ему следующие значения свойств:

```
RowSourceType = 2 - псевдоним;
```

```
RowSource = Graphs.title
```

В коде события `Init` для объекта `cmbData` запишем:

```
This.Value = Graphs.Title
```

В коде события `InteractiveChange` для объекта `cmbData`:

```
LOCAL cData, cCom
ThisForm.LockScreen = .T.
cData = This.Value
SELECT Graphs
LOCATE FOR Title = cData
cCom = Query
cCom = LEFT(cCom, LEN(cCom) - 2)
&cCom
ThisForm.RefreshGraph
ThisForm.ChangeGraphType
ThisForm.olbGraph.Object.ChartTitle.Caption = cData
ThisForm.LockScreen = .F.
```

Для выбора типа графика разместим в форме раскрывающийся список с именем `cmbType`. Присвоим ему следующие значения свойств:

- `RowSourceType = 5` - массив;
- `RowSource = aType` - имя массива;
- `BoundColumn = 2` - значение свойства `Value` определяется по второй колонке массива.

В коде события `Init` объекта `cmbType` для первоначального выбора первого типа графика запишем:

```
This.Value = 1
```

В коде события `InteractiveChange` для объекта `cmbType` предусмотрим выполнение пользовательского метода:

```

ThisForm.ChangeGraphType    Опишем для формы пользовательский метод ChangeGraphType:
LOCAL NType
nType = ThisForm.cmbType.Value
IF ThisForm.opgShow.Value = 1
    * Двухмерный график
    ThisForm.olbGraph.Type = NType
ELSE
    * Трехмерный график
    ThisForm.olbGraph.Type = NType + 8
ENDIF

```

Для выбора двухмерного или трехмерного вида графика разместим в форме две кнопки выбора в группе кнопок выбора `opgShow`.

В коде события `InteractiveChange` для объекта `opgShow` предусмотрим выполнение пользовательского метода:

```

ThisForm.ChangeGraphType
    Опишем для формы пользовательский метод RefreshGraph:
LOCAL cGraphString, nY_lim
cGraphString = ""
* Сколько полей должно участвовать в построении графика
nY_lim = Graphs.Y_number + 1
SELECT Datafile
* Создаем список полей, разделенных через табулятор,
* в конце строки записываем символ CHR(13)
FOR iCounter = 1 TO nY_lim
cGraphString = cGraphString + FIELDS(iCounter) ;
    + IIF(iCounter << nY_lim, CHR(9), CHR(13))
ENDFOR
* Приводим к символьному типу данные
* и формируем строку для передачи в OLE-объект
SCAN
    FOR iCounter = 1 TO nY_lim
        cGraphString = cGraphString + IIF(TYPE(FIELDS(iCounter))='C', ;
            EVALUATE(FIELDS(iCounter));
            ,str(EVALUATE(FIELDS(iCounter))));
        + IIF(iCounter << nY_lim, CHR(9), CHR(13))
    ENDFOR
ENDSCAN
SELECT Graphs
APPEND GENERAL graph DATA cGraphString

```

Построение графиков с помощью MS Excel 7.0

Работу с объектами Excel проиллюстрируем на примере создания формы для анализа данных в виде графиков, функционально аналогичной рассмотренной выше. При использовании Excel мы не связаны такими жесткими ограничениями на работу с данными, как в MS Graph, за счет того, что в Excel есть доступ к данным, на основании которых строится график через объект `Sheet`. В MS Graph данные располагаются в `DataSheet`, но к ним можно обеспечить только интерактивный, а не программный доступ. В то же время использование возможностей Excel требует больше ресурсов компьютера и лицензионной копии пакета на компьютере пользователя, в то время как MS Graph поставляется в составе всех пакетов разработки Microsoft. Посмотрите оба варианта форм для вывода графиков и выберите для себя более подходящий.

Для воплощения нашей идеи в Конструкторе формы создадим следующие объекты.

- `oleChart` - это объект `OLE Container Control`, с помощью которого мы встраиваем график Excel в форму. Нажимаем на панели инструментов `Form Controls` кнопку `OLE Container Control` и обводим в форме контур будущего графика. На экран выводится диалоговое окно `Insert Object`, в котором мы выбираем объект `Excel Chart`. Нажимаем кнопку `OK` и в обведенном контуре появляется график Excel, построенный на основании принятых по умолчанию данных, расположенных на втором рабочем листе таблицы. Одновременно, в соответствии с принципом редактирования на месте, изменяется главное меню `Visual FoxPro` для предоставления нам возможности отредактировать внешний вид представленного графика. Впоследствии можно вернуться к интерактивному

редактированию графика, нажав на этом объекте правую кнопку мыши и выбрав в контекстном меню команду **Edit**.

- **cmdExit** - кнопка управления для выхода из формы. Свойству **Caption** присвоим значение "OK".
- **cmdType** - кнопка управления для изменения типа графика. При нажатии на кнопку будет происходить изменение типа графика. Свойству **Caption** присвоим значение "Тип".
- **cmdSubType** - кнопка управления для изменения подтипа графика. При нажатии на кнопку будет происходить изменение подтипа графика. Свойству **Caption** присвоим значение "Подтип".
- **cmbData** - комбинированный список для изменения данных для графика. Для этого объекта установим следующие значения свойств, отличных от принятых по умолчанию, после того как разместим элемент управления в форме:

```
BoundColumn = 2
ColumnCount = 2
ColumnWidths = 150,0
FirstElement = 1
Name = cmbData
NumberOfElements = ALEN(aDataTables)
RowSource = aDataTables
RowSourceType = 5
Style = 2
Value = "Total1"
```

Как видно из перечисленных свойств, для формирования данных в списке мы будем использовать двумерный массив **aDataTables**. Первую колонку массива отведем для списка данных, доступных для графического отображения. Именно этот список будет видеть пользователь. Во второй колонке разместим названия источников данных (таблиц или курсоров). При запуске формы в списке будет выбран первый пункт, и для построения графика будет использована таблица **Total1**.

Остальные элементы управления не являются обязательными и служат для улучшения внешнего вида формы.

Для описания массива, используемого в комбинированном списке в событии **Load** объекта **Form**, запишем:

```
PUBLIC aDataTables(3,2)
aDataTables(1,1) = "Результаты продаж"
aDataTables(2,1) = "Доход"
aDataTables(3,1) = "Прибыль"
aDataTables(1,2) = "Total1"
aDataTables(2,2) = "Total2"
aDataTables(3,2) = "Total3"
```

Этот массив можно описать в пользовательской программе и вне формы, и тогда набор данных, доступных для графического отображения, можно будет изменять. С этой целью в качестве источника данных для комбинированного списка можно использовать и таблицу с двумя полями, которую легче редактировать, чем массив.

Для изменения данных на графике в соответствии с изменением выбранного пункта в списке в событии **InteractiveChange** объекта **cmbData** запишем следующий код:

```
ThisForm.LockScreen = .T.
SELECT (ThisForm.cmbData.Value)
&& Устанавливаем рабочую область
nRow = 2
SCAN && Считываем данные
ThisForm.oleChart.Object.Parent.Sheets("Sheet1").;
Cells(nRow,2).;
    Value = Sum1
ThisForm.oleChart.Object.Parent.Sheets("Sheet1").;
Cells(nRow,3).;
    Value = Sum2
ThisForm.oleChart.Object.Parent.Sheets("Sheet1").;
Cells(nRow,4).;
    Value = Sum3
```

```

nRow = nRow + 1
ENDSCAN
ThisForm.LockScreen = .F.

```

Здесь мы использовали блокировку изменения данных в форме, чтобы предотвратить последовательное изменение данных в каждом столбце графика (первая и последняя строка фрагмента кода). После ссылки на объект `oleChart` необходимо использовать свойство `Object` для того, чтобы `Visual FoxPro` понял, что указанное далее свойство `Parent` относится к объекту `Excel`. В противном случае `Visual FoxPro` решит, что мы ссылаемся на объект `Form` - родительский объект для объекта `oleChart`. Свойство `Parent` для объекта `Excel` нам необходимо для того, чтобы сослаться на объект `Workbook`. Только после этого мы можем использовать метод `Sheets` для получения ссылки на рабочий лист `Sheet1`, на котором находятся данные. Метод `Cells` позволяет нам указать конкретную ячейку таблицы, в которой мы хотим поменять данные с помощью свойства `Value`. Значения `Sum1`, `Sum2` и `Sum3` - это поля в таблицах, которые используются как источники данных для построения графиков и отображают изменение какого-то показателя во времени.

Для отображения данных из первой таблицы при запуске формы в событии `Init` объекта `Form` следует поместить такой код:

```

ThisForm.cmbData.Value = ThisForm.cmbData.List(1,2)
ThisForm.cmbData.InteractiveChange

```

Добавим в форму собственные свойства `NType` и `nSubType` для задания типа и подтипа выводимого графика. Для этого достаточно в меню `Form` выбрать команду `New Property`. По умолчанию присвоим им значения 3 и 1 соответственно.

В событии `Click` для объекта `cmdType` запишем следующий код:

```

IF ThisForm.nType > 13
    ThisForm.nType = 0
ENDIF
ThisForm.nType = ThisForm.nType + 1
ThisForm.OleChart.Type = ThisForm.nType

```

Это обеспечит приращение значения для типа графика при нажатии на кнопку "Тип" и сброс значения на начало отсчета при исчерпании числа типов графика.

В событии `Click` для объекта `cmdSubType` запишем следующий код:

```

IF ThisForm.nSubType > 3
    ThisForm.nSubType = 0
ENDIF
ThisForm.nSubType = ThisForm.nSubType + 1
ThisForm.OleChart.SubType = ThisForm.nSubType

```

Для некоторых типов графиков в `Excel` существует меньшее число подтипов или их вообще не существует. В этом случае при нажатии на кнопку "Подтип" изменений в графике не произойдет. При необходимости можно учесть эти нюансы, слегка модернизировав приведенный выше фрагмент.

В событии `Click` для объекта `cmdExit` для закрытия формы запишем следующий код:

```

RELEASE ThisForm

```

В завершение нам осталось включить в объект `DataEnvironment` формы таблицы, которые мы будем использовать в качестве источника данных для графика.

Внешний вид формы приведен на рис. 10.7.

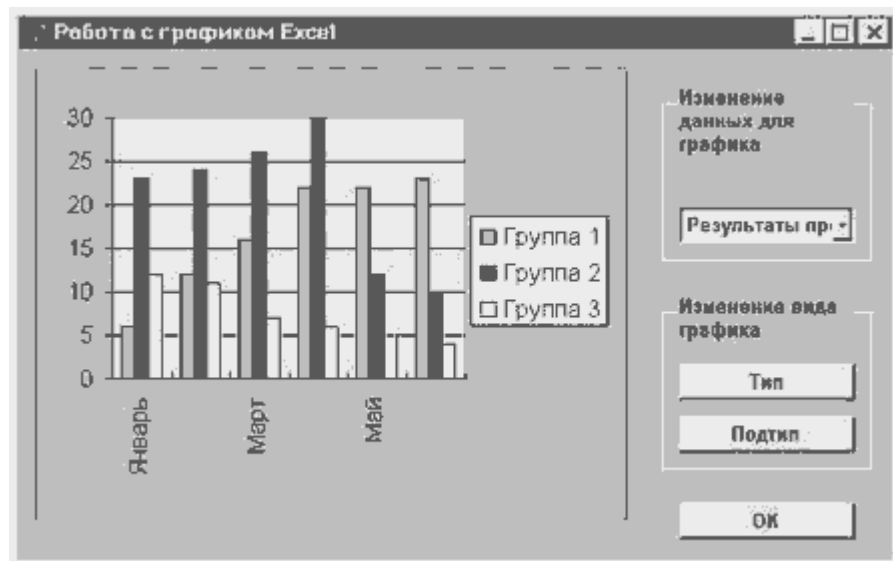


Рис. 10.7. Форма для представления графиков с помощью MS Excel 7.0

В современных условиях все больше специалистов могут самостоятельно использовать стандартные пакеты программ. Например, готовить графики и другую расчетную информацию с помощью пакета Excel. В этом случае достаточно только правильно импортировать данные из СУБД в Excel. Один из возможных способов представлен ниже. Он реализован в форме "Характеристика модели автомобиля", разработка которой была описана в [предыдущей главе](#).

Программный код для передачи данных в Microsoft Excel:

```
oleExcel=CREATEOBJECT("Excel.Application")
&&Создаем объект Excel
oleExcel.Visible=.T. && Делаем его видимым
oleExcel.Workbooks.Add && Добавляем Книгу
oleExcel.Selection.ColumnWidth = 20 && Ширина 1 колонки
oleExcel.Range("B1").Select
oleExcel.Selection.ColumnWidth = 12 && Ширина 2 колонки
oleExcel.Range("C1").Select
oleExcel.Selection.ColumnWidth = 12 && Ширина 3 колонки
oleExcel.Columns("A:A").Select && Отмечаем 1 колонку
oleExcel.Selection.WrapText = .T.
&& Перенос непоместившихся символов на следующую строку
oleExcel.Range("A1:C1").Select
oleExcel.Selection.HorizontalAlignment = 7
&& Заголовок размещаем в 1, 2 и 3 ячейках первой строки
oleExcel.Range("A2:C2").Select
oleExcel.Selection.HorizontalAlignment = -4108
&& Центрируем 1, 2 и 3 ячейки второй
&& строки
oleExcel.Columns("A:B").Select
oleExcel.Selection.Borders(2).Weight=2 && Линия справа
oleExcel.Columns("A:C").Select
oleExcel.Selection.Borders(3).Weight=2 && Нижняя линия
oleExcel.Range("A2:C2").Select
oleExcel.Selection.Borders(3).Weight=-4138
&& Верхняя линия
oleExcel.Selection.Borders(4).Weight=-4138
&& Нижняя линия
oleExcel.Range("A1:C1").Select
oleExcel.Selection.Font.Bold = .T.
oleExcel.Selection.Font.ColorIndex = 25
oleExcel.ActiveCell.FormulaR1C1 = "Характеристика модели автомобиля"
oleExcel.Range("A2").Select
oleExcel.ActiveCell.FormulaR1C1 = "Наименование модели"
oleExcel.Range("A2:C2").Select
oleExcel.Selection.Font.Bold = .T.
oleExcel.Range("B2").Select
```

```

oleExcel.ActiveCell.FormulaR1C1 = "Мощность"
oleExcel.Range("C2").Select
oleExcel.ActiveCell.FormulaR1C1 = "Крутящий момент"
***** Добавления значений *****
FOR I=1 TO KOL_Z
oleExcel.Cells(i+2,1).Value = all_value(i,1)
oleExcel.Cells(i+2,2).Value = all_value(i,2)
oleExcel.Cells(i+2,3).Value = all_value(i,3)
ENDFOR
***** Строим диаграмму *****
oleExcel.ActiveSheet.ChartObjects.Add(245, 10, 340, 270).Select
oleExcel.ActiveChart.;
chartwizard(oleExcel.range(oleExcel.cells(2,1),oleExcel.cells(12,3)),;
-4100,5,1,1,1,2,"Десять лучших","", "", "")
&& Где:
&& Gallery - -4100,
&& Format - 5,
&& PlotBy - 1,
&& CategoryLabels - 1,
&& SeriesLabels - 1,
&& HasLegend - 2,
    && Title - "Десять лучших",
    && CategoryTitle - "",
    && ValueTitle - "",
    && ExtraTitle - ""
***** Изменяем угол просмотра *****
oleExcel.ActiveChart.Elevation = 0
oleExcel.ActiveChart.Rotation = 328

```

Результат выполнения данного кода представлен на рис. 10.8.

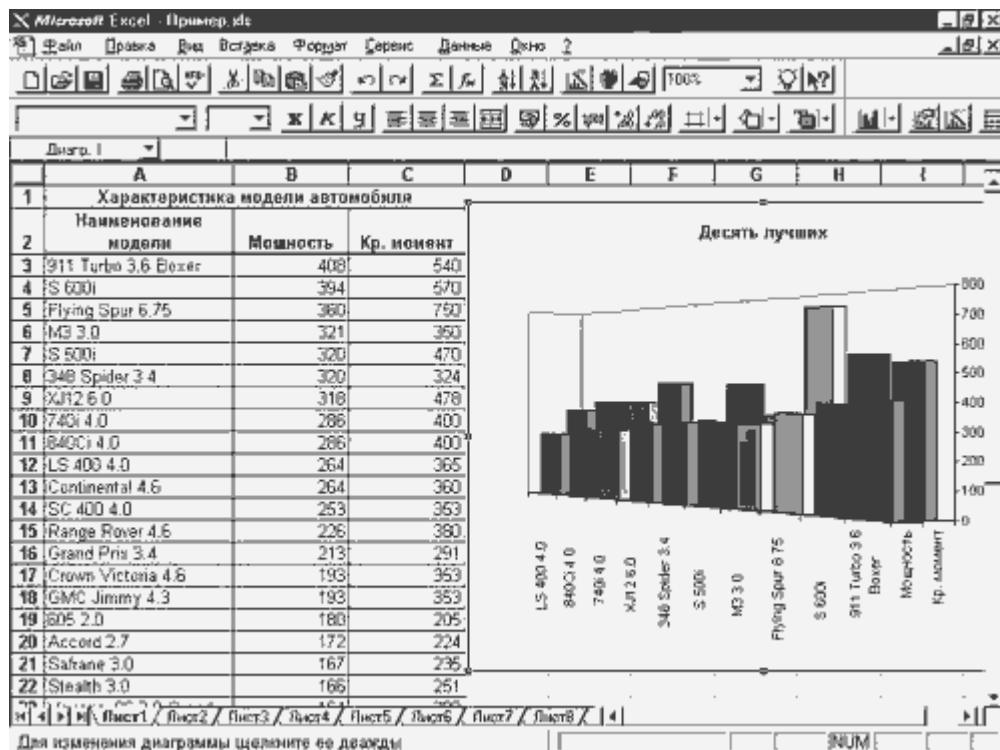


Рис. 10.8.

Построение отчета в Word for Windows

Чрезвычайно популярным пакетом для подготовки текстовой информации в нашей стране является текстовый процессор Word for Windows. Поэтому вполне объяснимо желание пользователя увидеть нужные ему данные в любимом редакторе и самостоятельно преобразовать

их в требуемый вид. Приведем образец кода для передачи данных из формы "Характеристика модели автомобиля" в Microsoft Word:

```
oleWord=CREATEOBJECT("Word.Basic")
oleWord.FileNewDefault
oleWord.AppShow
oleWord.FilePageSetup(,,"2,54 см","2,54 см","2,5 см","1,5 см","0 см","21 см","29,7 см")
oleWord.FormatFont("14",0,9,0,0,0,0,1,0,"0 пт","0 пт",0,"",,"Times New Roman Cyr",,1)
oleWord.FormatParagraph("0 см","0 см","0 пт","0 пт",0,"",1,1,0,0,0,0,0,"0","0 см")
oleWord.Insert("Характеристика модели автомобиля")
oleWord.InsertPara
oleWord.InsertPara
*****Создание таблицы*****
oleWord.TableInsertTable(,"3",ALLT(STR(Kol_z+1)))
oleWord.TableSelectColumn && Выделение колонки
oleWord.LeftPara && Выравнивание по левому краю
oleWord.TableSelectTable && Выделение таблицы
oleWord.TableRowHeight("0",2,"25 пт","0 см",0,1)
oleWord.TableSelectColumn && Выделение колонки
oleWord.LineDown(1)
oleWord.LineUp(1)
oleWord.TableSelectColumn && Выделение колонки
oleWord.TableColumnWidth("11 см","0,38 см") && Установка ширины выделенной колонки
oleWord.NextCell
oleWord.NextCell
oleWord.TableSelectColumn && Выделение колонки
oleWord.TableColumnWidth("3,5 см","0,38 см")
oleWord.NextCell
oleWord.NextCell
oleWord.TableSelectColumn && Выделение колонки
oleWord.TableColumnWidth("3,5 см","0,38 см")
oleWord.TableSelectTable && Выделение таблицы
oleWord.FormatBordersAndShading(3,0,2,2,2,2,1,1,0,0,0,0,0,0,"0 пт",0,0,0,"0",-1)
oleWord.CharRight(1)
oleWord.LineUp(1)
oleWord.TableSelectRow && Выделение ряда
oleWord.TableHeadings
oleWord.Bold
oleWord.CenterPara
oleWord.FormatParagraph("","","","","4","25 пт")
oleWord.CharLeft(1)
oleWord.Insert("Наименование модели")
oleWord.NextCell
oleWord.Insert("Мощность")
oleWord.NextCell
oleWord.Insert("Крутящий момент")
oleWord.NextCell
***** Добавления значений в таблицу*****
FOR I=1 TO KOL_Z
    oleWord.Insert(all_value(i,1))
    oleWord.NextCell
    oleWord.Insert(STR(all_value(i,2)))
    oleWord.NextCell
    oleWord.Insert(STR(all_value(i,3)))
    IF I<<KOL_Z
        oleWord.NextCell
    ENDIF
ENDFOR
```

Результат выполнения данного кода представлен на рис. 10.9.

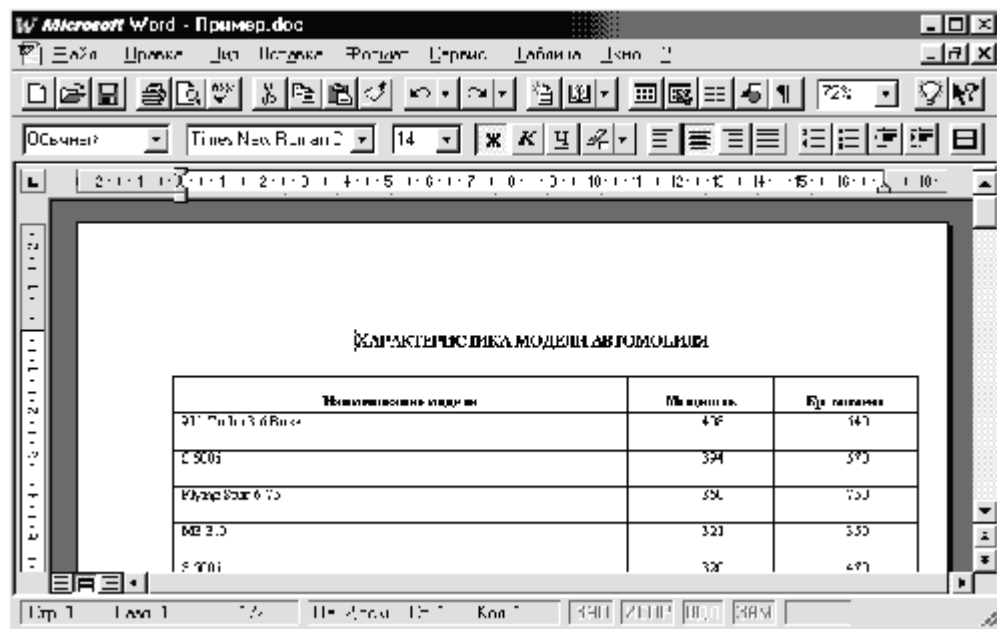


Рис. 10.9. Пример передачи данных в Microsoft Word

Запись информации в Schedule+

Используя возможности OLE Automation и наличие в составе пакета Microsoft Office специальной программы для планирования - Schedule+, мы можем расширить возможности нашего приложения за счет включения в него функций бизнес-планирования.

Microsoft Schedule+ является средством бизнес-планирования, которое может использоваться в локальном режиме и в режиме совместного доступа. Программа Schedule+ является OLE-сервером, и поэтому мы можем использовать объекты этой программы для хранения, планирования и управления сведениями о встречах, собраниях, задачах, контактах и событиях. При планировании в расписание заносится оповещение, что позволяет не забыть о важной встрече, задаче или событии.

Организация объектов в Schedule+ несколько отличается от Excel или MS Graph. Здесь имеются два типа объектов: таблица и пункт. Каждая таблица состоит из определенного количества строк, которые и представляются пунктами. Каждый пункт имеет определенный набор свойств. Описанная структура приведена на рис. 10.10.

Объект типа *таблица*

Пункт 1
Пункт 2
Пункт 3
...

Объект типа *пункт*

Свойство 1	Свойство 2	...
------------	------------	-----

Рис. 10.10. Структура объектов в Schedule+

В качестве объектов типа таблица рассматриваются основные функциональные возможности Schedule+. Это запланированные события (Appointments), записи о лицах (Contacts), предупреждения (Alarms) и т. д. В свою очередь каждое событие, запись и т. п. является объектом типа пункт и имеет набор свойств. Например, для того чтобы записать сведения о новом лице в Visual Basic, необходимо выполнить такую программу:

```
Sub NewContact()
```

```

Dim oSchedApp As Object, oSchedTable As Object, oSchedItem As Object
* Запускаем скрытую копию Schedule+
Set oSchedApp = CreateObject("SchedulePlus.Application")
* Проводим процедуру регистрации
If Not oSchedApp.LoggedOn Then
oSchedApp.LogOn
End If
* Устанавливаем ссылку на объект типа таблица.
* Свойство ScheduleLogged возвращает объект планирования для
* зарегистрированного пользователя.
Set oSchedTable = oSchedApp.ScheduleLogged.Contacs
* Устанавливаем ссылку на объект типа пункт (новый пункт в таблице)
Set oSchedItem = oSchedTable.New
* Записываем в пункт данные с помощью его свойств
oSchedItem.SetProperties FirstName:="Андрей", LastName:="Горев", _
Notes:="Эффективная работа с СУБД на основе решений Microsoft", _
PhoneBusiness:="(812)259-4277", PhoneFax:="(812) 112-6872"
` Стираем ссылки на объекты
Set oSchedItem = Nothing
Set oSchedTable = Nothing
Set oSchedApp = Nothing
End Sub

```

После выполнения этой программы откройте Schedule+, и вы увидите, что на вкладке Contacts появилась новая запись.

Совершенно аналогично можно записать в Schedule+ какое-либо планируемое событие. В этом случае дата и время начала и окончания планируемого события должны быть указаны обязательно.

```

Sub NewAppoint()
Dim oSchedApp As Object, oSchedTable As Object, oSchedItem As Object
Set oSchedApp = CreateObject("SchedulePlus.Application")
If Not oSchedApp.LoggedOn Then
oSchedApp.LogOn
End If
' Устанавливаем ссылку на таблицу планируемых событий
Set oSchedTable = oSchedApp.ScheduleLogged.Appointments
' Создаем новое событие, в котором хотим участвовать
Set oSchedItem = oSchedTable.New
' Описываем это событие
oSchedItem.SetProperties Text:="DevCon97", _
Notes:="Ежегодная международная конференция разработчиков Microsoft", _
Start:="(06/10/97 10:00)", _
End:="(06/13/97 18:00)"
' Стираем ссылки на объекты
Set oSchedItem = Nothing
Set oSchedTable = Nothing
Set oSchedApp = Nothing
End Sub

```

10.4. Применяем ActiveX

В этом параграфе на примерах иерархического списка и календаря мы опишем, как можно использовать элементы ActiveX в пользовательском приложении.

Иерархический список

Одним из поставляемых компонентов ActiveX (OCX) для средств разработки Microsoft является Outline. Необходимо сказать, что элемент управления Outline (файл MSOUTL32.OCX) представляет собой особую разновидность списка, в котором можно отображать элементы в иерархическом порядке. Этим пользуются при схематическом изображении каталогов и файлов в файловой системе. Именно такой метод применен в Windows 95 и Windows NT 4.0.

У каждого элемента в списке Outline могут быть подчиненные элементы, которые визуальн

представляются дополнительными уровнями с отступами. Когда элемент разворачивается, его подчиненные элементы становятся видимыми. Когда элемент сворачивается, его подчиненные элементы скрываются. Элементы в списке **Outline** могут также сопровождаться графикой, служащей визуальным обозначением состояния элемента.

Элемент списка может сопровождаться любыми графическими элементами из числа следующих:

- **Линии древовидной структуры.** Вертикальные и горизонтальные линии, связывающие первичные элементы с подчиненными. Линия дерева генерирует события **Expand** и **Collapse**.
- **Отступ.** Характеристика уровня подчиненности элемента. Каждый уровень отступа соответствует определенному уровню подчиненности, который вы задаете с помощью свойства **Indent**.
- **Значок "плюс/минус".** Указывает, видимы подчиненные элементы или скрыты. Если щелкнуть мышкой на значке "плюс", подчиненные элементы становятся видимыми и значок "плюс" заменяется значком "минус". Если щелкнуть мышкой на значке "минус", подчиненные элементы скрываются и значок минус заменяется значком "плюс".
- **Изображения типа.** Отображают состояние элемента. На рисунках с изображением типа обычно показывается, допускает ли элемент, обладающий подчиненными элементами, разворачивание или свертывание. Например, изображение закрытой папки указывает на то, что элементы каталога можно развернуть. Состояние элемента определяется пользователем. Рисунок с изображением типа генерирует события **PictureClick** и **PictureDbClick**.
- **Текст.** Символьная строка, отображаемая для элемента.

Каждый графический элемент может представлять собой область, которая способна реагировать на действия пользователя. Если щелкнуть на изображении элемента, активизируется специальная группа событий.

Свойство **Style** позволяет задать или запретить отображение всех или некоторых графических элементов для каждого элемента списка **Outline**.

Чтобы выбрать элемент списка, необходимо щелкнуть (или дважды щелкнуть) на строке с соответствующим текстом. Щелчка мышкой только на графическом элементе недостаточно.

В качестве примера рассмотрим создание формы для Администратора БД, с помощью которой он может проконтролировать доступ пользователей к полям и таблицам БД. Для отображения структуры данных в БД **Auto_Store** используем элемент управления **Outline**.

Итак, начнем с того, что в **Visual FoxPro** создадим новую форму. Затем, выбрав команду **Options** из меню **Tools**, активизируем вкладку **Controls**. На данной вкладке в списке **Selected** установим крестик на пункте **Outline Control**, как это показано на рис. 10.11. Далее нажмем кнопку **View Classes** панели инструментов **Controls** и в открывшемся меню выберем **OLE Controls**. На рис. 10.12 показано меню кнопки **View Classes**. Таким образом, у вас появится возможность визуально поместить данный элемент управления в форму посредством нажатия кнопки **Outline Control** (рис. 10.13). Однако в **Visual FoxPro** есть и более простой способ подключения элементов управления **ActiveX**, включающий в себя следующие действия:

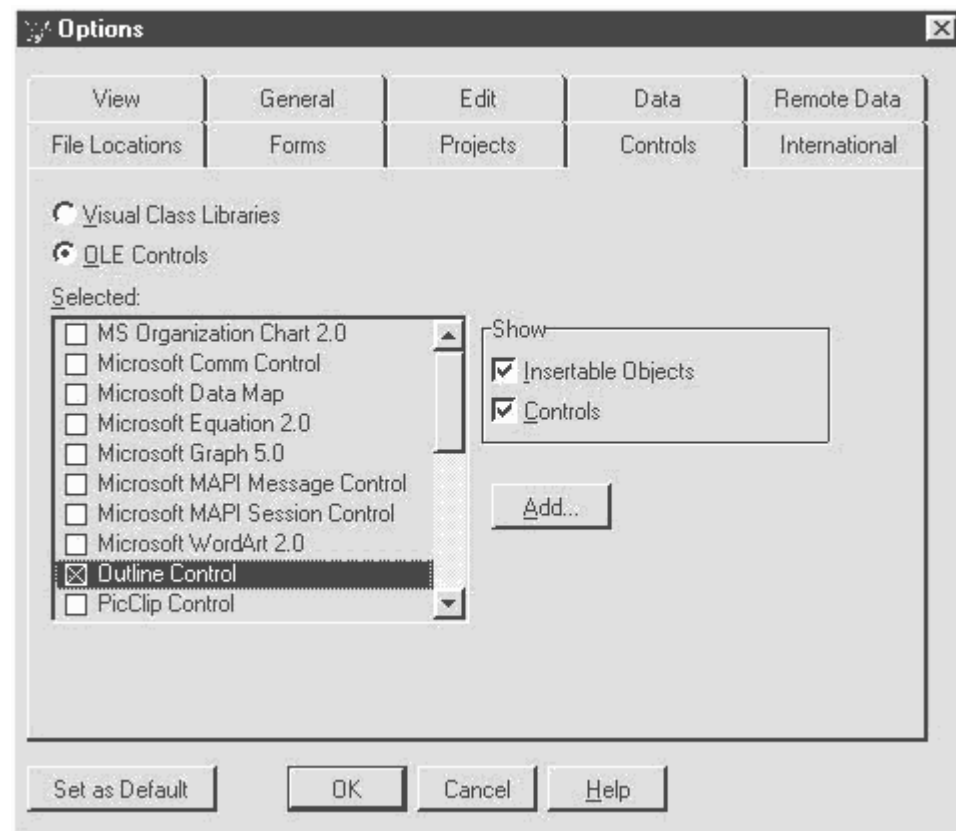


Рис. 10.11.



Рис. 10.12.

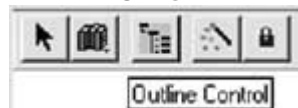


Рис. 10.13.

1. Создаем новую форму.
2. Нажимаем кнопку **OLE Container Control** панели инструментов **Controls**.
3. Обводим мышкой контур на форме для элемента управления **Outline Control**.
4. В открывшемся диалоговом окне **Insert Object** выбираем кнопку **Insert Control** и в списке **Control Type** дважды щелкаем по элементу **Outline Control** (рис. 10.14).

Независимо от выбранного варианта действия в форме появится новый объект (рис. 10.15). Для определения свойств объекта **Outline** нажмите правую кнопку мыши на данном объекте и из контекстного меню выберите команду **Properties**.

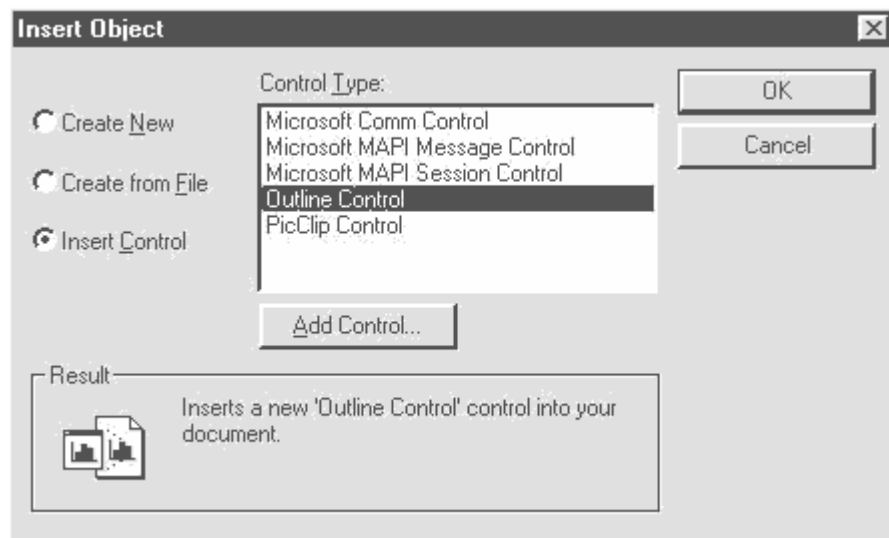


Рис. 10.14. Диалоговое окно Insert Object для размещения объекта Outline Control

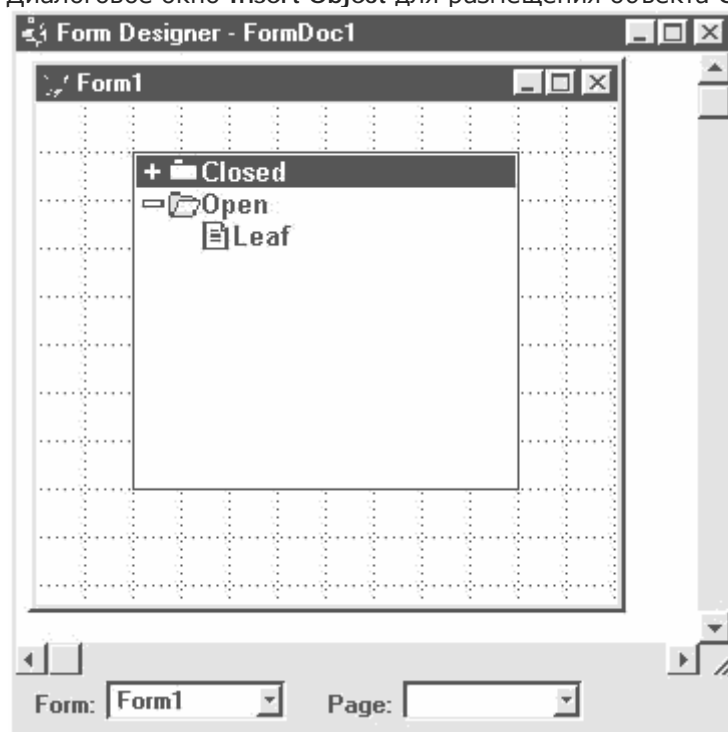


Рис. 10.15. Элемент управления Outline в разрабатываемой форме

Для определения собственных изображений в иерархическом списке в диалоговом окне Properties активизируем вкладку Pictures, где для свойств PictureClosed и PictureOpen установим соответствующие графические файлы. Найти их на диске можно с помощью диалогового окна, вызываемого при нажатии кнопки Browse (рис. 10.16).

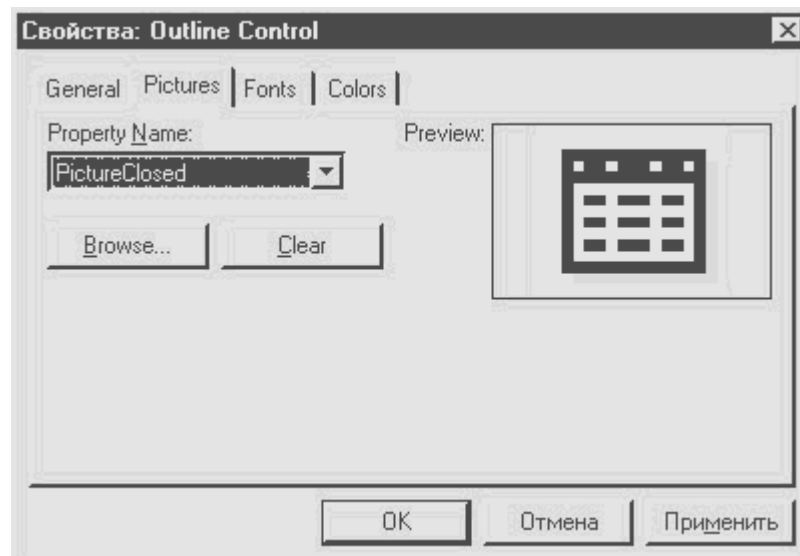


Рис. 10.16. Вкладка Pictures в диалоговом окне свойств Outline Control

Для определения цвета фона (BackColor) и цвета текста (ForeColor) в диалоговом окне Properties активизируем вкладку Colors, где для свойства ForeColor определяем синий цвет (рис. 10.17).

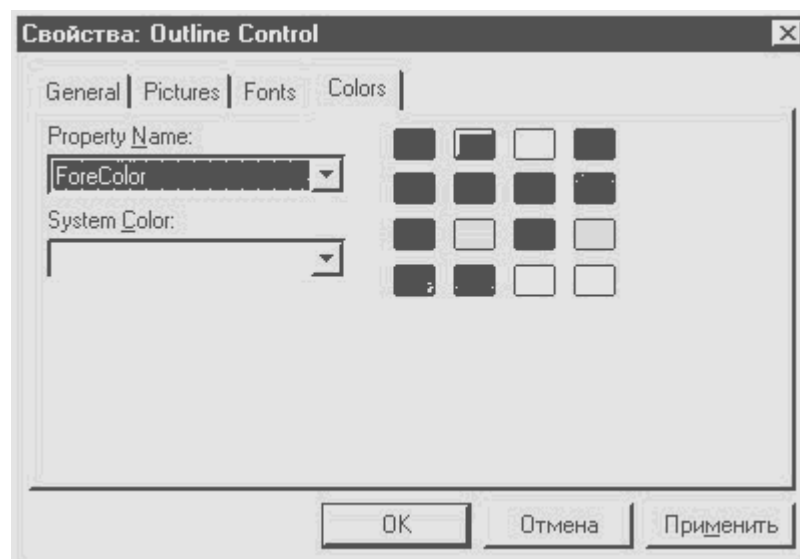


Рис. 10.17.

На рис. 10.18 показан Конструктор формы, использующий элемент управления Outline Control и заданные свойства ForeColor, PictureClosed и PictureOpen.

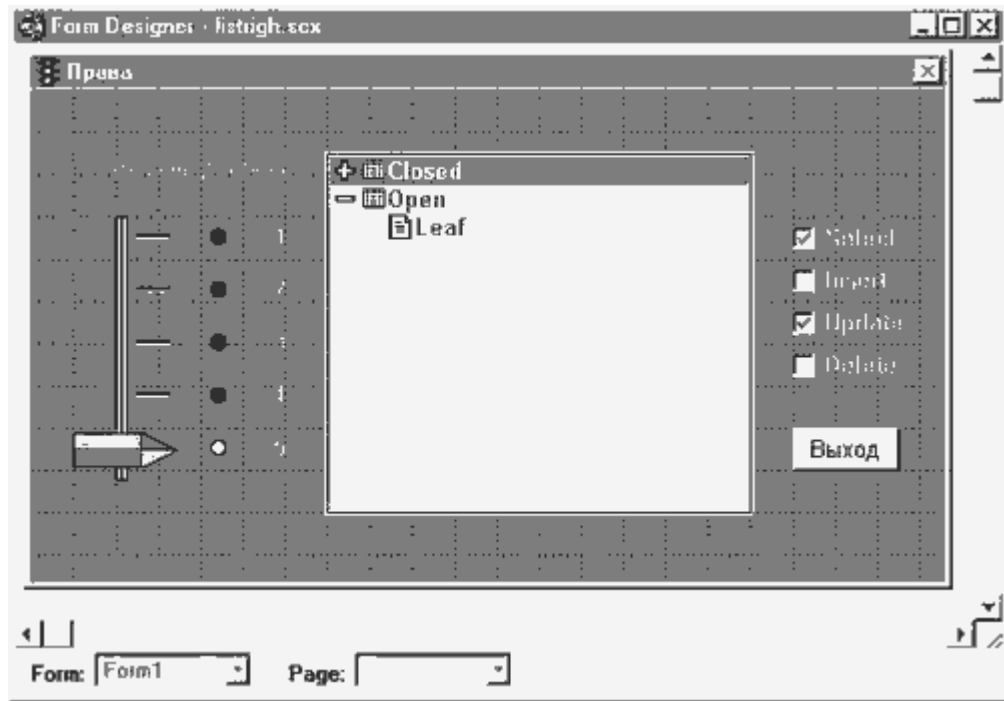


Рис. 10.18.

Для события Init объекта Olec1 записываем следующий код:

```

&& Помещаем в массив aTabList список таблиц из словаря данных.
&& В связи с установленным уровнем доступа, равным по умолчанию пяти,
&& данный список таблиц фильтруем с помощью предложения WHERE
SELECT tableslist.table_name ;
FROM datavocabulary!tableslist ;
WHERE ALLT(Tableslist.table_name)<<>>"Sale" ;
AND ALLT(Tableslist.table_name)<<>>"Account" ;
AND ALLT(Tableslist.table_name)<<>>"Customer" ;
AND ALLT(Tableslist.table_name)<<>>"Order" ;
AND ALLT(Tableslist.table_name)<<>>"Salesman" ;
INTO ARRAY aTabList
&& Определяем значение переменной nnPabl
nnPabl=0
&& Последовательно заносим список полей для каждой таблицы
&& в массивы 1nnAr, 2nnAr, 3nnAr, <193>
&& После чего в объект Olec1 добавляем имена таблиц и соответствующих
&& этим таблицам полей, а для каждого поля устанавливаем отступ (indent)=2
&& и тип изображения (PictureType)=2
FOR X=1 TO ALEN(aTabList)
  ThisForm.Olec1.AddItem(ALLT(aTabList(X)))
  nnTab=ALLT(aTabList(X))
  nnAr="nnAr"+ALLT(STR(X))
  SELECT Tables.field_name ;
  FROM datavocabulary!tables ;
  WHERE Tables.table_name = nnTab ;
  INTO ARRAY &nnAr
  nnAddItem="ThisForm.Olec1.AddItem"
  nnArn="nnAr"+ALLT(STR(X))
  FOR I=1 TO ALEN(&nnAr)
    &nnAddItem(ALLT(&nnAr(i)))
    nnBeby=nnPabl+X+I-1
    ThisForm.Olec1.indent(nnBeby)=2
    ThisForm.Olec1.PictureType(nnBeby)=2
  ENDFOR
  nnPabl=nnPabl+ALEN(&nnAr)
ENDFOR

```

Остальные элементы формы не имеют значения для работы иерархического списка, и мы не будем на них останавливаться. Готовая запущенная форма показана на рис. 10.19.

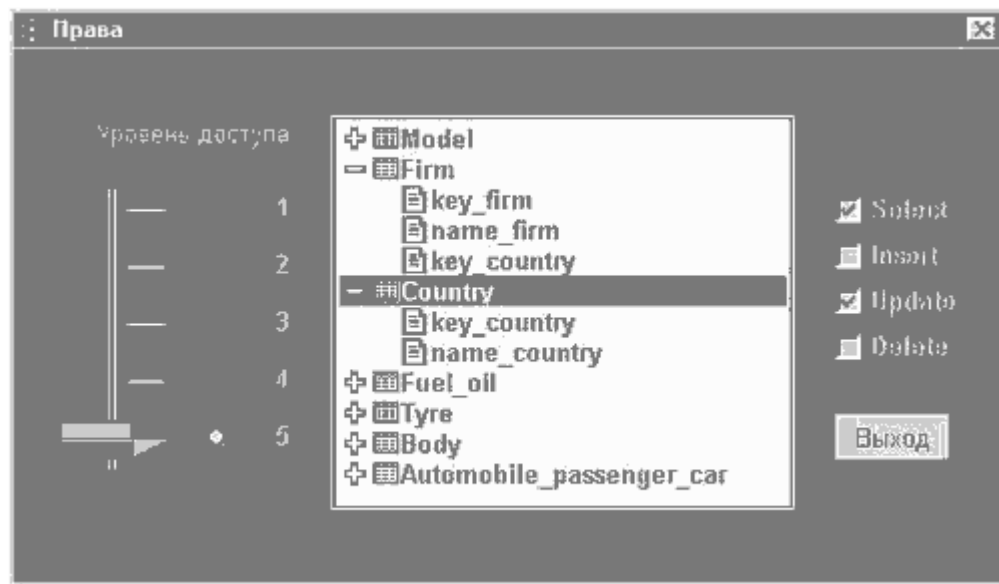


Рис. 10.19. Форма с иерархическим списком в действии

Календарь

В следующем примере рассмотрим использование календаря для ввода или редактирования данных в поле, содержащем дату. Вместо набора даты вручную пользователь должен иметь возможность визуального выбора в календаре требуемой даты. После выбора дата должна автоматически записаться в нужное поле, как это показано на рис. 10.20. Для этого в Visual Basic откроем новый проект и создадим несложную форму. Чтобы сделать процесс создания формы максимально простым, используем, пожалуй, единственный визуальный инструмент разработчика, присутствующий в Visual Basic, - Data Form Designer - Конструктор формы для работы с данными, показанный на рис. 10.22. Необходимые пояснения для построения формы вы найдете на этом же рисунке. Построенная Конструктором форма приведена на рис. 10.21. Как видите, она мало пригодна для использования в пользовательском приложении, хотя черновая работа по размещению полей и выбора для них элементов управления выполняется автоматически без нашего участия. Мы привнесли в форму минимальные изменения и за счет корректировки значений свойств привели ее к виду, показанному на рис. 10.20. Потребовалось выполнить для объектов Label и CommandButton изменения значений свойства Caption, перемещение и изменение размеров некоторых элементов управления. Если у вас возникнут трудности с изменением размера элемента управления Data Control, установите значение свойства Align равным 0.

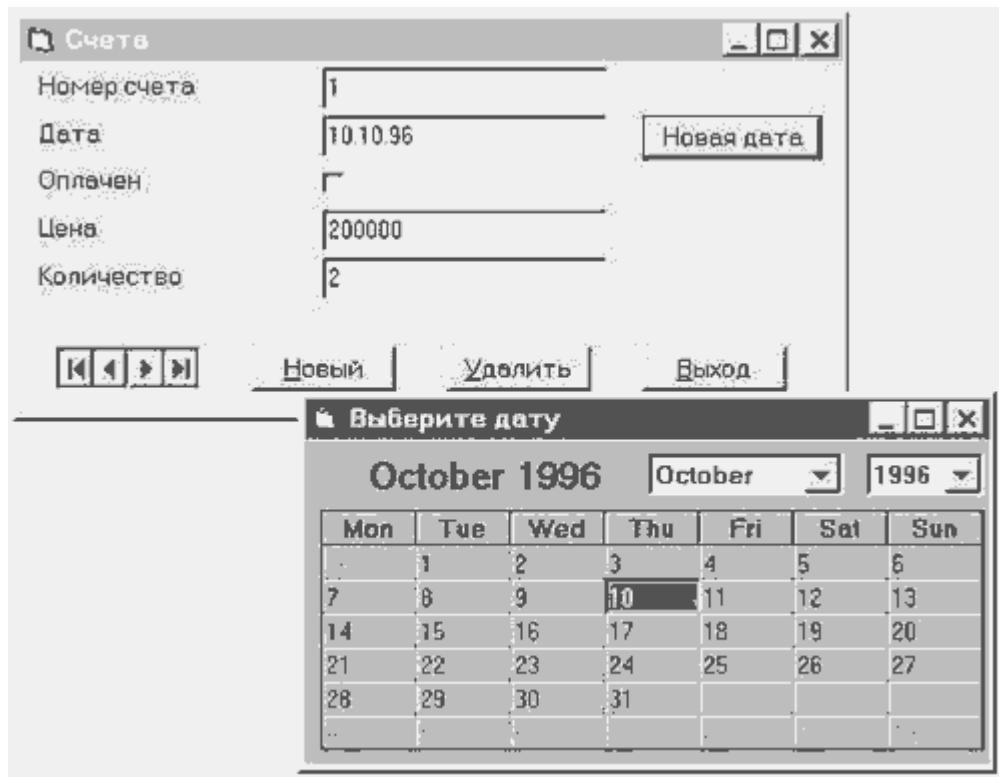
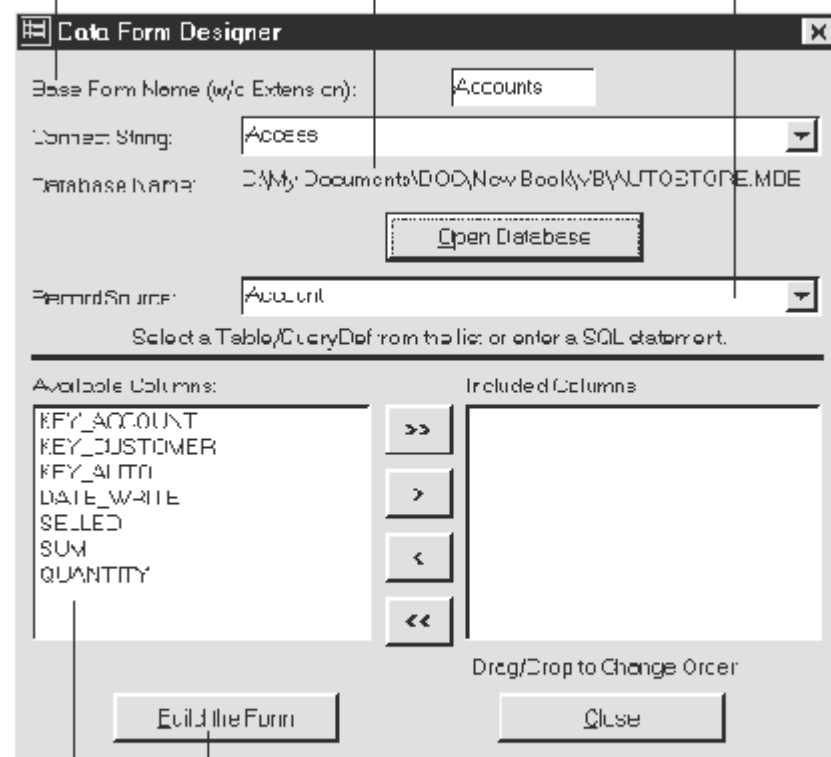


Рис. 10.20. Изменение даты с помощью календаря

Наименование таблицы (или представления) Наименование создаваемой формы

Источник данных



Создать форму

Исходный список полей в таблице Список полей, размещаемых в форме

Рис. 10.21.

Рис. 10.22.

Для вызова календаря разместим рядом с полем для отображения даты еще одну кнопку. Дадим ей имя `cmdCallData` и заголовок "Новая дата". Щелкнем на кнопке два раза и в появившемся окне для записи кода в процедуре `cmdCallData_Click()` запишем строку:

```
frmNewDate.Show
```

Тем самым мы обеспечиваем вызов формы с календарем.

Для создания вспомогательной формы с календарем в меню *Insert* выберем команду *Form*. Дадим ей имя `frmNewDate` и заголовок "Выберите дату". На панели инструментов нажмем кнопку *Calendar* и, удерживая левую кнопку мыши, обведем на форме соответствующий контур. Появившемуся на форме объекту дадим имя `olcCalendar`. В *Visual Basic* свойства для объекта *ActiveX* можно изменять и в его собственном окне свойств, вызываемом из контекстного меню, и в окне *Properties Visual Basic*. Изменить значение даты, записанное в поле первой формы, можно с помощью следующей процедуры:

```
Private Sub olcCalendar_AfterUpdate()
    frmAccounts.txtFields(1) = olcCalendar.Value
End Sub
```

Теперь, при запуске формы и изменении даты в календаре, тут же будет изменяться значение поля `date_write` в таблице `Account`.

Глава 11

Подготовка отчетных данных

11.1. Создание отчетов в Visual FoxPro

Управление режимом печати

11.2. Создание отчетов в Access

Подготовка и оформление отчетных данных, без сомнения, являются кульминацией любого делового приложения. Именно на этом этапе ваш заказчик наконец-то узнает, что вы делаете и зачем он нанял вас на работу. Результат этого знания не обязательно будет сопровождаться только восторженными репликами в ваш адрес. Так что стоит приложить усилия для того, чтобы все многообразие данных, записанных в таблицах приложения, нашло достойное отображение в отчетах, напечатанных принтером.

11.1. Создание отчетов в Visual FoxPro

Visual FoxPro имеет мощные средства построения отчетов для вывода данных в желаемом для пользователя виде на принтер, экран или в текстовый файл.

В этом параграфе мы рассмотрим последовательность создания отчета в *Visual FoxPro*, способы его печати и управления режимами печати отчета.

Перед началом построения отчета необходимо продумать его расположение на странице и порядок вывода данных. Сделайте эскиз отчета на листе бумаги, учтите общую ширину всех полей, которые вы хотите распечатать, подумайте о длине заголовков. Таким образом вы получите представление о требуемом формате отчета, только не забудьте о реальных возможностях вашего принтера.

Порядок расположения данных в отчете и его элементы (для примера его длина взята равной трем страницам) приведены на рис. 11.1. Данные, помещаемые в полосу Details, распечатываются многократно, с автоматическим переводом указателя записи в таблице или курсоре, которые являются источником данных. Группировка данных позволяет выводить их в систематизированном виде. Например, список сотрудников по отделам, отделы по цехам и т. п.

Конструктор отчета (Report Designer) по принципу работы похож на Конструктор формы без учета объектно-ориентированных возможностей. На рис. 11.2 приведен вид Конструктора отчета и даны пояснения его основным элементам.

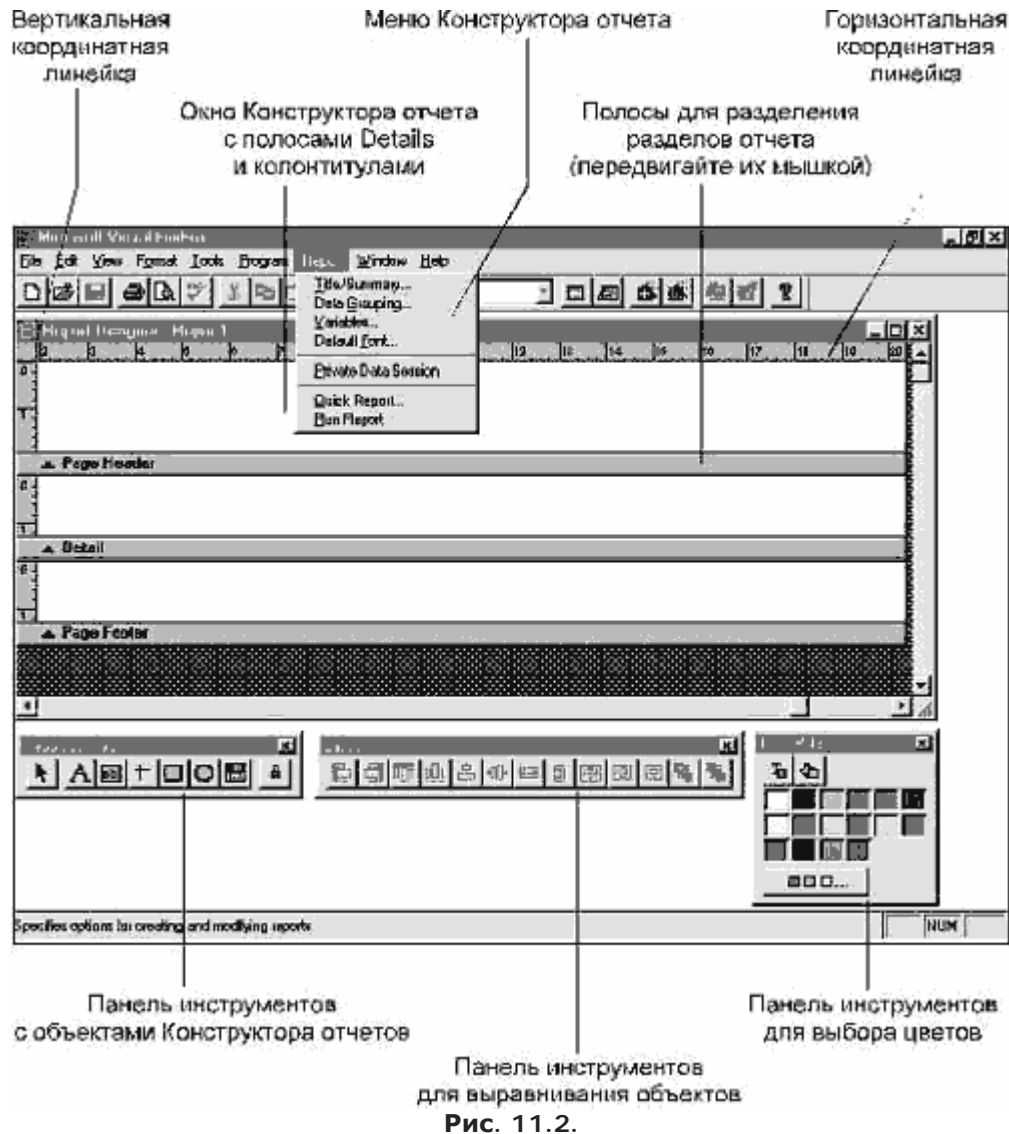


Рис. 11.2.

С помощью меню Конструктора отчета *Report*, которое автоматически появляется в главном меню Visual FoxPro, можно выполнить следующие действия:

- *Title/Summary* - добавить в окно Конструктора отчета полосы заголовка и итогов.
- *Data Grouping* - вывести диалоговое окно для задания группировки данных.
- *Variables* - задать переменные для использования при подсчете данных.
- *Default Fonts* - вывести стандартное диалоговое окно Windows для выбора шрифта и установки его характеристик, которые будут использованы по умолчанию для всех текстовых объектов в отчете.

- **Private Data Session** - установить для отчета отдельную сессию данных. В этом случае перемещение указателя записи при печати отчета не будет сказываться на форме, в которой используются те же данные.
- **Quick Report** - запустить утилиту быстрого построения отчета.
- **Run Report** - запустить отчет на выполнение.

Большое количество действий можно выполнить также с помощью команд, расположенных в меню **View** и **Format**, основные из которых мы рассмотрим при описании последовательности создания отчета.

Для ускорения процесса работы с объектами при создании отчета можно пользоваться панелями инструментов, которые также показаны на рис. 11.2. О двух из них мы уже говорили при описании Конструктора формы. Панель инструментов **Report Controls** по своим функциям похожа на панель инструментов **Form Controls** в Конструкторе формы и предназначена для выбора объектов отчета. Назначение кнопок на этой панели описано на рис. 11.3.

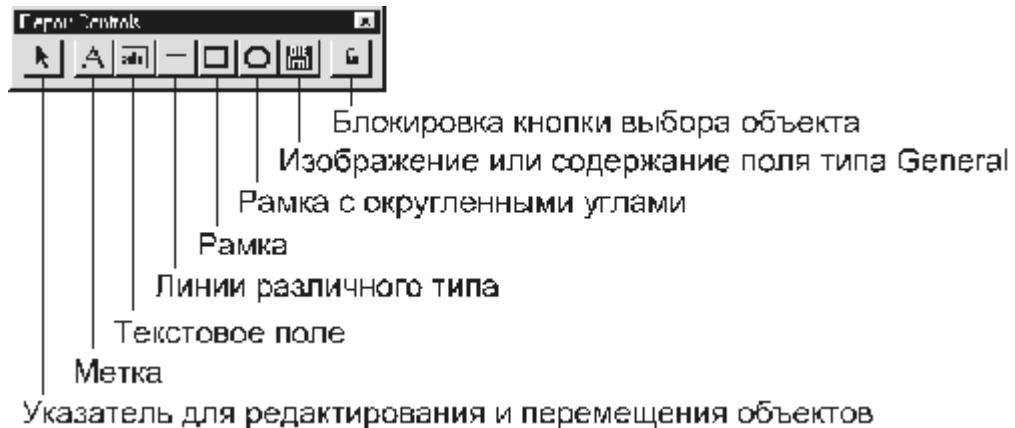


Рис. 11.3. Панель инструментов Report Controls

Простейший отчет по данным из одной таблицы удобно строить с помощью команды **Quick Report** из меню **Report**. Она автоматически помещает выбранные поля в окно Конструктора отчетов. После ее выбора появляется диалоговое окно **Quick Report**, изображение которого приведено на рис. 11.4.

Расположение полей по строкам

Расположение полей по колонкам

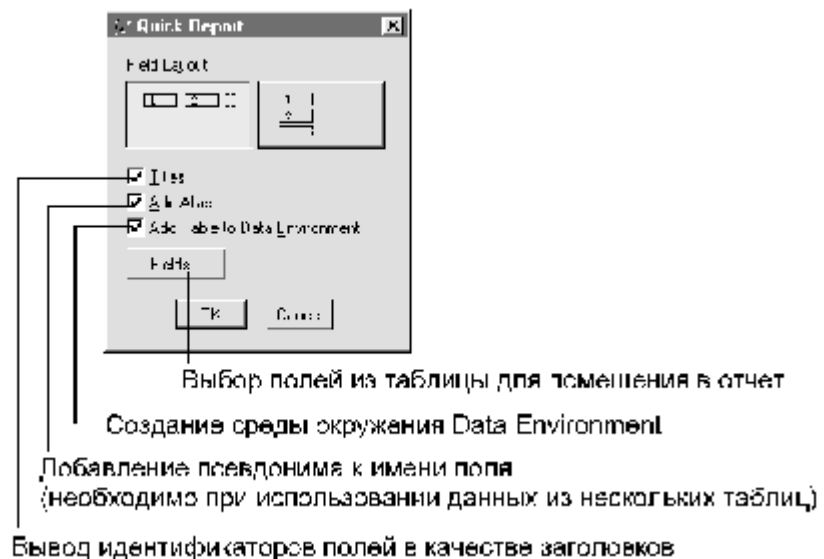
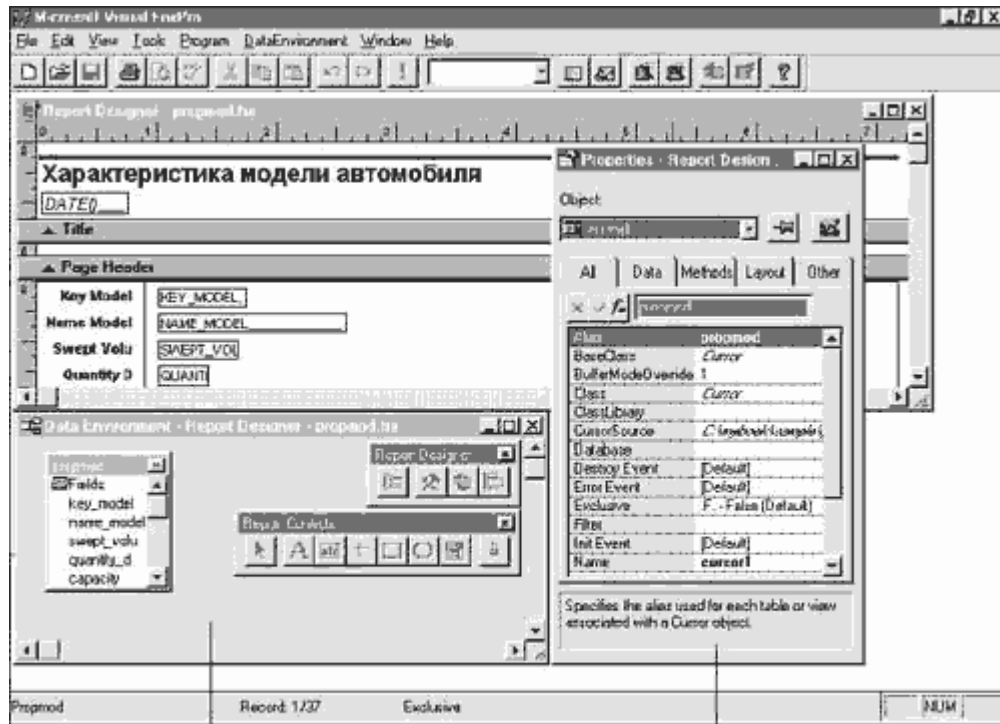


Рис. 11.4.

По умолчанию команда **Quick Report** помещает все выбранные поля в полосу **Details**, а идентификаторы полей - в полосу **Page Header**. В полосу **Page Footer** слева помещается поле с функцией **DATE()** для вывода текущей даты, а справа - поле с системной переменной **_PAGE NO** (номер страницы) и меткой "Page" перед ним. Если вам нужна простая распечатка данных, то это неплохая заготовка, в которой останется только поменять идентификаторы полей на их

заголовки и, прибавив к отчету полосу Title, оформить его название. Кстати, для подготовки отчетов можно гораздо шире, чем это было в случае создания формы, использовать Мастер создания отчета, так как объекты отчета не привязаны к классам.

На рис. 11.5 приведен отчет, построенный с помощью команды **Quick Report** для таблицы **Propmod**, в которой хранятся данные о моделях автомобилей. Перед снятием этого изображения с экрана компьютера мы вызвали команду **Data Environment** меню **View**. Как видите, мы точно так же, как это было в Конструкторе формы, можем задавать свойства и реагировать на события, связанные с данными, используемыми в отчете. Наличие среды окружения позволяет легко использовать при построении отчета специально подготовленные с помощью команды **SELECT-SQL** данные. Достаточно создать просмотр требуемой структуры и затем просто поместить его в среду окружения - **Data Environment**.



Окно среды окружения Data Environment

Окно Properties для установки свойств среды окружения

Рис. 11.5.

Заметьте, что при проектировании отчета мы можем широко применять технологию перетаскивания. Для быстрого размещения полей в отчете их можно перетаскивать как из окна **Data Environment**, так и из **Project Manager**. Также можно перетаскивать таблицы из **Project Manager** в окно **Data Environment**.

Как вы, наверное, уже поняли из сказанного выше, отчеты в **FoxPro** состояются из объектов, которыми можно манипулировать множеством способов. В число этих объектов входят графические объекты (линии и рамки), объекты полей (поля, переменные, выражения и т. п.) и текстовые объекты. Только не запутайтесь в различных нюансах слова "объект" в различных инструментальных средствах **Visual FoxPro**. В Конструкторе отчета объекты, размещаемые на его поверхности, в отличие от Конструктора формы никакого отношения к объектно-ориентированному программированию не имеют.

Все действия в Конструкторе отчета производятся только с выделенными объектами. Для выделения объекта установите на нем указатель и нажмите кнопку мыши. Можно выделить более одного объекта, для этого, выделив первый объект, нажмите клавишу **Shift**, после чего выделяйте другие объекты. Несколько объектов можно выделить также с помощью маркера выбора (selection marquee). Установите курсор вне тех объектов, которые должны быть выбраны. Нажмите клавишу пробела или кнопку мыши, чтобы закрепить маркер выбора. Вместо курсора появится точка, начиная с которой при перемещении курсора строится прямоугольник вокруг тех объектов, которые должны быть выделены. Если вы работаете мышью, не отпускайте ее кнопку. Когда все объекты будут выделены, нажмите клавишу **Enter** или отпустите кнопку мыши.

Если выделено несколько объектов, любая операция перемещения, вырезания, копирования, вставки ранее вырезанного объекта или удаления действует на них, как на один объект.

Изменения параметров объектов выполняются с помощью следующих команд меню **Format**:

- **Align** - выравнивание объекта или группы объектов относительно страницы отчета или друг друга. Используйте панель инструментов **Layout** для быстрого выполнения этой функции.
- **Size** - позволяет изменить размеры объекта или группы объектов относительно друг друга. Команды подменю соответствуют возможностям панели инструментов **Layout**. Команда **To Grid** изменяет размеры объекта в соответствии с установленной координатной сеткой, если включено **Snap to Grid**.
- **Horizontal Spacing** - позволяет увеличить (**Increase**), уменьшить (**Decrease**) или выровнять (**Make Equal**) промежутки между объектами по ширине страницы отчета.
- **Vertical Spacing** - позволяет увеличить (**Increase**), уменьшить (**Decrease**) или выровнять (**Make Equal**) промежутки между объектами по длине страницы отчета.
- **Bring to Front** - позволяет выдвинуть объект на передний план, если он закрыт другими объектами.
- **Send to Back** - позволяет отодвинуть объект на задний план, если он закрывает другой объект.
- **Group** - группирует выделенные объекты так, что в дальнейшем мы можем изменять их свойства и местоположение, рассматривая их как единый объект.
- **Ungroup** - разъединяет группу объектов для индивидуального задания свойств.
- **Snap to Grid** - включает или выключает режим привязки объекта к координатной сетке при его перемещении.
- **Set Grid Scale** - включает или выключает режим отображения координатной сетки.
- **Font** - выводит на экран стандартное диалоговое окно **Windows** для изменения типа и характеристик шрифта для меток и текстовых полей.
- **Text Alignment** - позволяет установить выравнивание текста в метках и текстовых полях. Для изображений и встроенных OLE-объектов доступно выравнивание только по центру. Для многострочных меток здесь же мы можем задать межстрочный интервал.
- **Fill** - позволяет выбрать заполнение для областей внутри рамок.
- **Pen** - позволяет установить ширину или тип линии или рамки.
- **Mode** - позволяет выбрать прозрачный или сплошной фон для всех объектов отчета кроме линии.

Новые объекты в отчете можно создать с помощью панели инструментов **Report Controls** (см. рис. 11.3). Если вы создаете текстовое поле, то после того, как вы обведете контур, в котором это поле будет располагаться, на экране появится диалоговое окно **Report Expression**, приведенное на рис. 11.6.

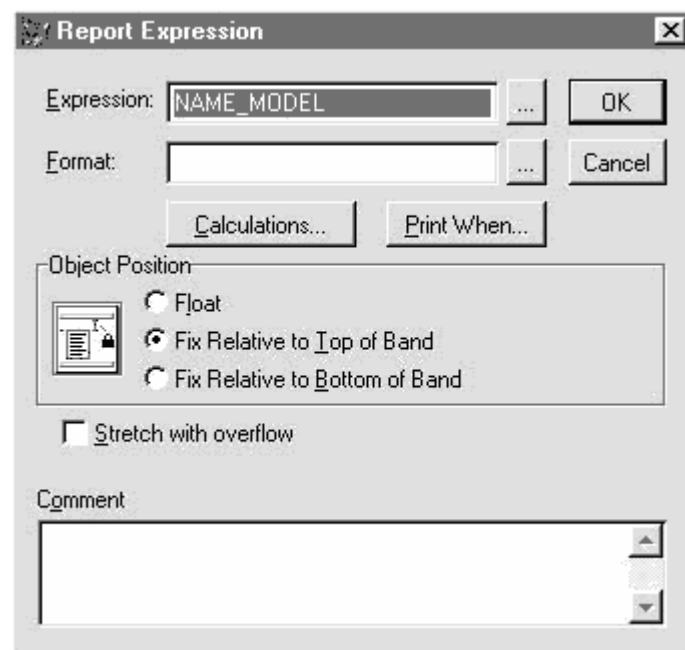


Рис. 11.6.

С помощью этого диалогового окна в отчет можно поместить изменяемые данные (поля) с определенным форматом вывода, которые могут быть полями таблицы, просмотра или курсора, выражениями, функциями, в том числе определенные пользователем.

Задать поле, которое необходимо поместить в отчет, можно прямо, указав его в поле

Expression диалогового окна, или вызвав Построитель выражений, нажав кнопку справа от этого поля. Пользуясь Построителем выражений, вы можете построить выражение, выбирая поля из открытых таблиц и размещенные в памяти переменные и стандартные функции Visual FoxPro в диалоговом режиме. Также возможно проверить правильность составленного выражения.

В поле Format диалогового окна Report Expression можно задать шаблон, который будет определять формат выводимого значения. Для определения формата можно использовать следующие символы:

- A - допускает вывод только символов алфавита.
- L - допускает вывод только логических данных.
- N - допускает вывод только букв и цифр.
- X - допускает вывод любых символов.
- 9 - допускает вывод только цифр для символьных данных и цифр и знаков для численных данных.
- # - допускает вывод цифр, пробелов и знаков числа.
- \$ - выводит знак доллара в фиксированной позиции перед числом.
- \$\$ - выводит знак доллара непосредственно перед числом.
- * - выводит знак "звездочка" перед числом для затруднения приписки дополнительных цифр в отпечатанном документе.
- . - определяет положение десятичной точки.
- , - может использоваться для отделения разрядов в числе.

Например, если для числового поля мы установим шаблон "9,999,999.99", то в выводимом значении в указанных местах будут использованы символы, принятые в качестве разделителей разрядов и десятичной точки.

Кнопка справа от поля Format поможет вам выбрать подходящий формат в диалоговом режиме. В табл. 11.1 приведен список опций, которые можно задать в диалоговом режиме в поле Format для полей каждого типа.

Таблица 11.1. Форматы для полей в отчете

Опция	Пояснение
Для редактирования символьных данных	
To Upper Case	Все символы алфавита преобразуются в заглавные буквы
Ignore Input Mask	Несоответствующие формату символы в шаблоне не отображаются при печати
Left Justify	Данные печатаются в поле, выровненными влево
Right Justify	Данные печатаются в поле, выровненными вправо
Center Justify	Данные в поле центрируются
Для редактирования числовых данных	
Left Justify	Выравнивание данных по левой границе
Blank if Zero	Пропуск поля, если его значение равно 0
(Negative)	Отрицательные числа будут заключаться в скобки
CR if Positive	После положительного числа печатается CR - кредит
DB if Negative	После отрицательного числа печатается DB - дебит
Leading Zeros	В поле печатаются все ведущие нули
Currency	Отображает данные с добавлением наименования денежной единицы
Scientific	Число печатается в экспоненциальной форме
Для редактирования данных типа даты	

Edit "SET"	Данные редактируются как дата с учетом
Date	текущего формата даты, установленного командой SET DATE
British Date	Данные редактируются по европейскому стандарту даты

В блоке **Object Position** устанавливаются условия печати данных, в том случае, когда сами эти данные могут иметь совершенно различный объем. Включение кнопки выбора **Float** позволяет смещать точку начала печати данных в зависимости от места на странице отчета, в котором была закончена печать данных, расположенных выше. Обычно данные печатаются в пределах того контура, который отводится каждому полю. Если при проектировании отчета вы не уверены точно в объеме данных или для сокращения пустого пространства на странице нежелательно отводить для какого-то поля максимальный резерв пространства, то вы можете щелкнуть на поле проверки **Stretch with overflow**. Это позволит при необходимости продолжить печать данных на последующих строках в пределах установленной ширины поля. Вот в этом случае и важно для объектов отчета, расположенных ниже в той же полосе (например, линия для разделения записей), установить опцию **Float**.

Если поле проверки **Stretch with overflow** не включено, то место для печати данных из текущего поля останется фиксированного размера. Все данные из поля, которые не войдут в предназначенное для них пространство, будут отсечены.

Опция **Fix Relative to Top of Band** устанавливается по умолчанию и обеспечивает печать данных с начала поля.

Опция **Fix Relative to Bottom of Band** позволяет привязать данные к нижней границе поля.

При нажатии на кнопку **Calculate** на экране отобразится диалоговое окно, с помощью которого можно организовать выполнение определенных вычислений, перечень которых приведен ниже.

В верхней части диалогового окна **Calculate** находится раскрывающийся список **Reset**, с помощью которого можно устанавливать момент сброса значения поля в начальное значение. Имеющиеся в этом списке значения позволяют подсчитывать данные в целом по отчету, по странице, по группе данных или по колонке. Возможные виды вычислений, которые можно задать с помощью кнопок выбора в этом диалоговом окне, приведены в табл. 11.2.

Вторая кнопка, которая имеется в диалоговом окне **Report Expression**, - это **Print When**. В диалоговом окне с таким же названием, которое появляется при нажатии на эту кнопку, можно установить условия печати данных.

В блоке **Print Repeated Values** можно выбрать одну из двух кнопок выбора - **Yes** или **No**. Опция **Yes** установлена по умолчанию, в этом случае в отчете печатаются все значения текущего поля. Выбор опции **No** подавляет печать повторяющихся значений поля, кроме первого.

Поля проверки в блоке **Also Print** позволяют устанавливать условия печати при переносе данных на последующий лист или колонку.

Поле проверки **Remove Line If Blank** позволяет исключить из отчета пустые строки, если в расположенных на них полях отсутствуют данные.

С помощью текстового поля **Print Only When Expression is True** вы можете указать условие для печати данных из текущего поля.

С каждым созданным объектом можно связать примечания, которые никак не влияют на отчет, но могут служить напоминанием об объекте или, например, содержать фрагмент кода, который потом необходимо будет переписать в прикладную программу.

Таблица 11.2. Виды вычислений для полей отчета

Опция	Пояснение
Nothing	Никакие вычисления не будут выполняться над этим полем (по умолчанию)
Count	Подсчитывает, сколько раз данное поле печатается в группе, на странице или в отчете, в зависимости от выбора Reset
Sum	Вычисляет сумму значений поля нарастающим итогом
Average	Вычисляет среднеарифметическое (среднее) значение поля в группе, на странице или в отчете
Lowest	Отображает наименьшее значение этого поля для группы, страницы или отчета
Highest	Выводит наибольшее значение поля
Std. Deviation	Возвращает квадратный корень из дисперсии для значений переменной в

группе, на странице или в отчете

Variance Это статистическая характеристика, измеряющая степень отклонения конкретного значения поля от среднего по всем этим значениям в группе, столбце, странице или отчете

Управлять процессом печати отчета можно и с помощью специальных событий, которые формируются в начале и в конце печати каждой полосы отчета. Реакцию на эти события можно установить в диалоговом окне, которое возникает при двойном щелчке на полосе, разделяющей разделы отчета в окне Конструктора отчета. Вид этого окна приведен на рис. 11.7.

Предотвращает сжатие за счет пустых строк
или увеличения при печати установленной высоты полосы



Рис. 11.7.

Компоновка страницы устанавливается командой *Page Setup* из меню *File*. Основные элементы диалогового окна *Page Setup* приведены на рис. 11.8. Обратите внимание, что именно здесь можно установить условия для создания многоколончатого отчета.

Выполнить **группировку данных в отчете** для их сортировки по какому-либо признаку позволяет команда *Data Grouping* из меню *Report*. Допускается до 20 уровней группировки.

Для правильной группировки данные в таблице должны быть либо отсортированы, либо проиндексированы по признаку группировки.

С каждой группой можно выполнить следующие операции:

- выполнение вычислений над записями внутри заданной группы;
- печать текста в верхних и нижних колонтитулах;
- переход на новую страницу перед началом печати каждой группы;
- установка номера страницы в начальное состояние при печати групп с новой страницы.

Для создания группы данных необходимо выполнить следующие действия:

1. Выберите команду *Data Grouping*. Появится диалоговое окно, приведенное на рис. 11.9.
2. Введите групповое выражение, которое будет определять признак смены группы, в поле *Group Expression*. С помощью расположенной справа кнопки можно вызвать Построитель выражений и сформировать групповое выражение.
3. Включите нужные опции в блоке *Group Properties* (см. рис. 11.9).

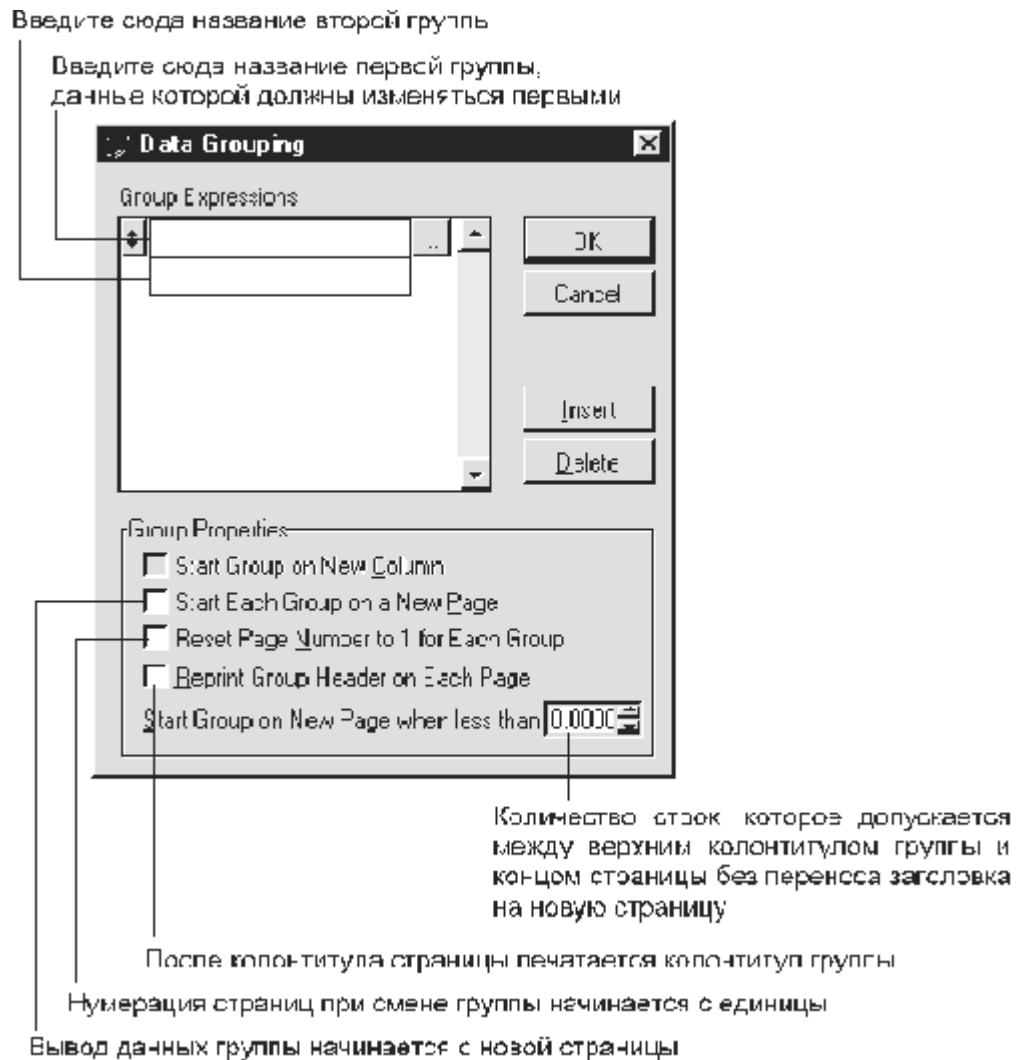


Рис. 11.9.

Для того чтобы добавить другие группы, повторите этот процесс. Группы перечисляются в списке Group Expression в порядке их создания. В окне Конструктора отчета имена полос групп содержат номера этих групп и усеченные групповые выражения. Верхние и нижние колонтитулы группы с меньшими номерами находятся ближе к полосе Details. Группа с меньшим номером основывается на выражении, значение которого в отчете изменяется реже, чем для группы с большим номером. Это значит, что группа с большим номером является подгруппой группы с меньшим номером.

Группировка данных может использоваться и с целью печати определенных данных на отдельных листах, например при распечатке счетов или накладных, которые, как правило, распечатываются на основе данных, содержащихся в одной записи. В этом случае в качестве признака группировки данных необходимо использовать номер записи, а в окне Data Grouping отметить поле проверки Start Each Group on a New Page.

С помощью команды *Variables* из меню *Report* можно определить **переменные в отчете**, которые будут использоваться в процессе его построения. Переменные удобно использовать для хранения промежуточных результатов вычислений, в качестве поля в отчете или как часть выражения. При выборе команды *Variables* на экране отображается диалоговое окно Report Variables, с помощью которого можно создать новую переменную, изменить существующую или убрать ее (рис. 11.10).

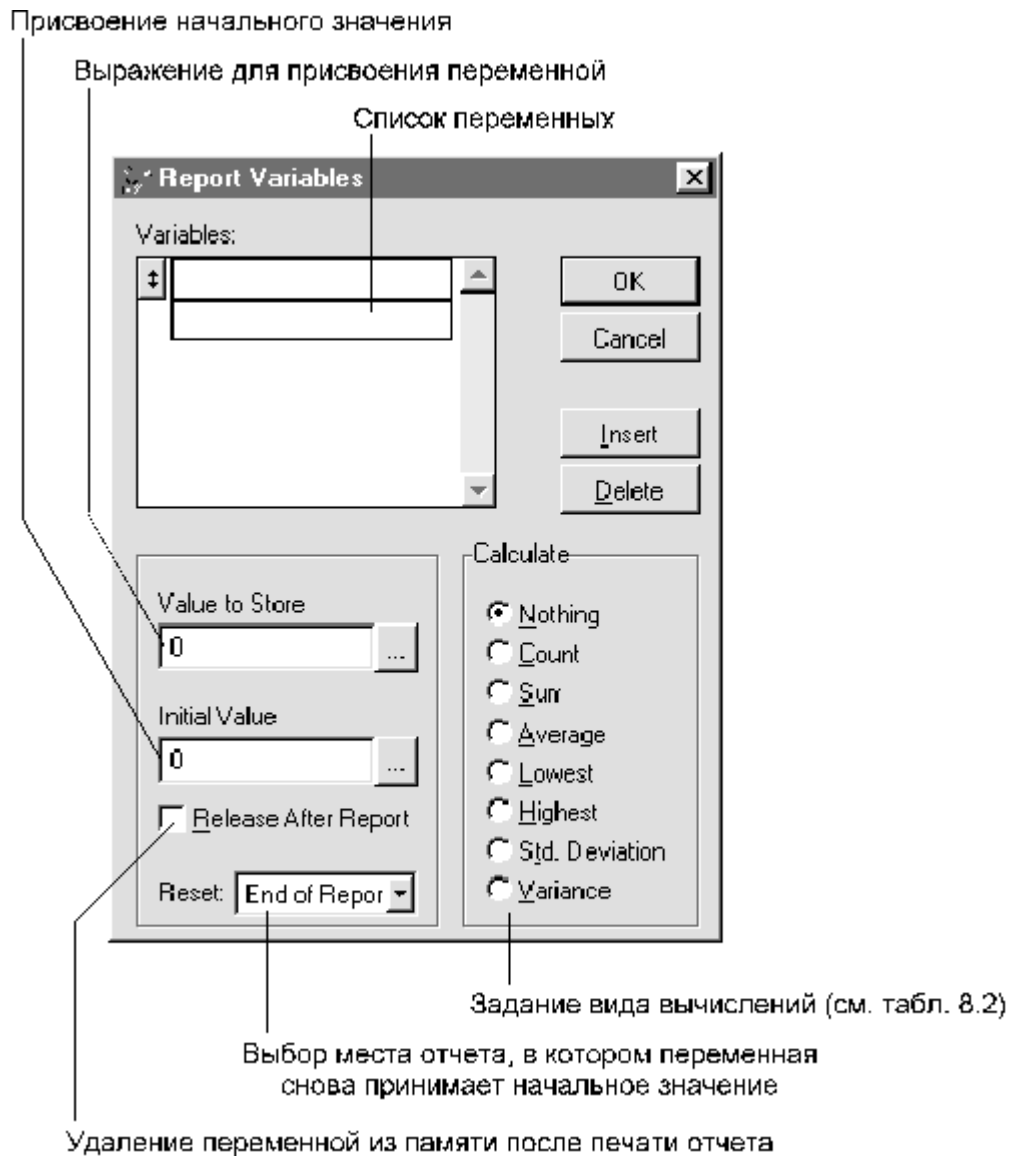


Рис. 11.10.

Значение переменной при вычислениях будет зависеть от задания диапазона ее действия, то есть от того, когда ее значение будет возвращаться в первоначальное.

После создания переменные доступны для создания любых выражений в отчете посредством Построителя выражений.

Последовательность, в которой переменные отображаются в списке **Variables**, может влиять на выходные данные. Переменные вычисляются в порядке их появления в списке. Если одна переменная используется для определения значения другой переменной, то первая переменная должна находиться в списке раньше второй.

С помощью переменных в отчете решаются и довольно нетривиальные задачи. Например, для печати итоговых значений через каждые десять записей мы можем в диалоговом окне **Report Variables** создать переменную. Назовем ее **nRec**. Выберем в блоке **Calculate** тип выполняемых вычислений - **Count** (подсчет числа значений). В диалоговом окне **Data Grouping** укажем в качестве признака группировки **INT(nRec/10)**. В окне Конструктора отчета появятся дополнительные полосы **Group Header** и **Group Footer**, в них мы можем разместить поля, для которых необходимо печатать итоговые значения.

Когда отчет готов, можно проверить результат, выбрав команду **Preview** в меню **View**. Для управления предварительным просмотром используйте специальную панель инструментов, с помощью которой можно перемещаться по страницам, регулировать масштаб изображения, а потом вернуться в режим проектирования.

Мы уже неоднократно подчеркивали, что наиболее удобно и эффективно создавать отчет по заранее подготовленным данным. Для этого, например, можно использовать просмотры. Очевидно, что этот подход позволит в будущем избежать излишних переделок пользовательской программы в связи с переходом на технологию клиент-сервер - ведь в этом случае не имеет значения, где находятся те данные, которые вы используете для составления отчета. Необходимо

будет только внести соответствующие изменения в просмотр. При этом методика и техника составления отчета совершенно не зависят от типа источника данных.

Visual FoxPro предлагает разработчику большое число методов создания отчета. Рассмотрим последовательность создания несложного отчета, которую авторы успешно применяют на протяжении многих лет при разработке пользовательских приложений. Этот метод предусматривает:

1. Создание источника данных для отчета в виде, наиболее полно отвечающем структуре отчета. Это может быть запрос, представление или таблица. Главное, чтобы данные, необходимые для отчета, были сведены в источник данных перед непосредственным выполнением отчета, а не получались в процессе его печати из большого числа связанных таблиц.
2. Использование Мастера отчета для быстрого получения чернового варианта разрабатываемого отчета.
3. Доведение отчета в Конструкторе до законченного вида.

Для ускорения создания отчета используем Мастер отчета. В **Project Manager** выберем вкладку **Documents**, перейдем на заголовок **Reports** и дадим команду **New**. В появившемся диалоговом окне выберем кнопку **Report Wizard** и в следующем окне из трех вариантов Мастера отчета выберем самый простой - **Report Wizard**. На первом шаге Мастера выберем нужный источник данных, как это показано на рис. 11.11. На последнем шаге напомним для отчета заголовок и сохраним его под именем **PROPMOD.FRX** и загрузим для дальнейшей модернизации в Конструктор отчета. Мы увидим отчет в виде, представленном на рис. 11.12. Надо отметить, что отчет практически готов. Нам осталось навести внешний лоск: заменить кое-где английский текст и придать отчету солидный вид, украсив его, например, эмблемой фирмы. При необходимости ничто не препятствует и проведению некоторой перекомпоновки полей. В любом случае это можно сделать быстрее, чем размещать их заново.

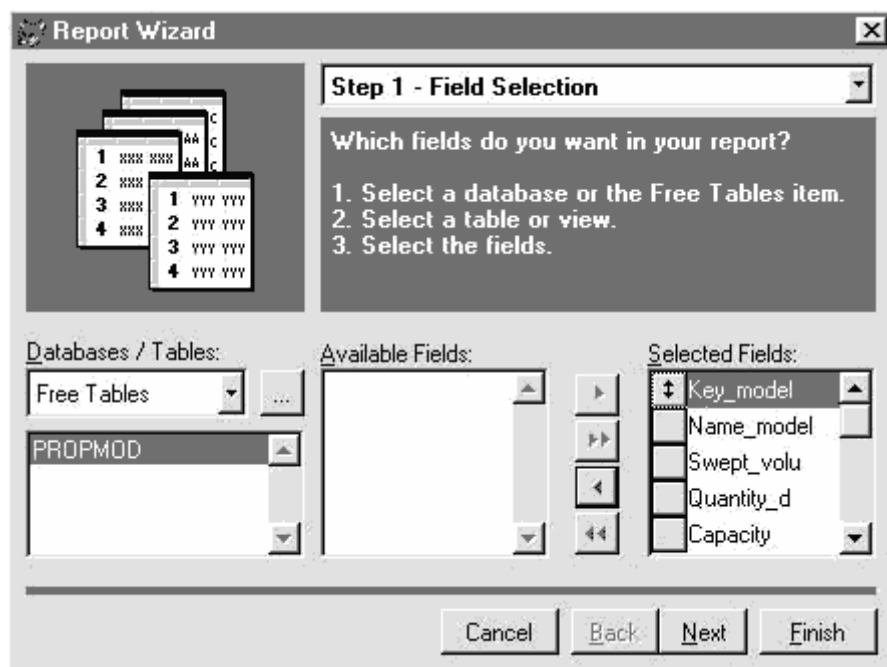


Рис. 11.11. Выбор источника данных в Мастере отчета

Report Designer - propmod.rpt

Характеристика модели автомобиля

DATE()

▲ Title

▲ Page Header

Key Model	KEY_MODEL
Name Model	NAME_MODEL
Swept Volu	SWEPT_VOL
Quantity D	QUANT
Capacity	CAPACITY
Torque	TORQUE
Top Speed	TOP_SPEED
Starting	STARTING
Length	LENGTH
Width	WIDTH
Height	HEIGHT
Expense 98	EXPENSE
Name Firm	NAME_FIRM
Name Count	NAME_COUNT
Name Fuel	NAME_FUEL

Рис. 11.12. Отчет PROPMOD в Конструкторе отчета после завершения работы с Мастером отчета

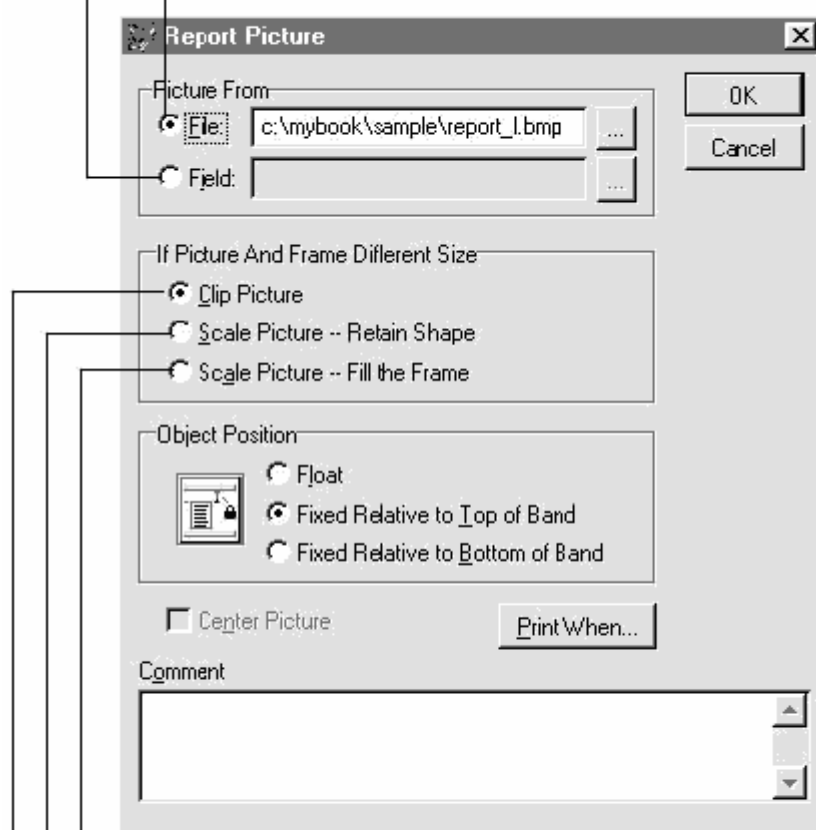
Для редактирования текста в метках нажмите кнопку Label на панели инструментов Report Controls и щелкните курсором на тексте.

Чтобы поместить в отчет изображение, нажмите кнопку Picture/OLE Bound Control на панели инструментов Report Controls и обведите контур в нужном месте отчета. Появится диалоговое окно, которое приведено на рис. 11.13. В нем надо указать изображение, которое вы хотите поместить в отчет. Три кнопки выбора в средней части диалогового окна позволяют установить способ, с помощью которого размер изображения будет приводиться в соответствие контуру, который вы обрисовали в отчете для размещения изображения. По умолчанию принят способ, при котором масштаб изображения не изменяется. В этом случае, если контур меньше изображения, оно не будет видно целиком и проблема может быть решена только за счет увеличения размера контура. Два других варианта предусматривают масштабирование изображения, определяемое размерами контура или пропорциями самого изображения. Очевидно, что в большинстве случаев предпочтительным будет вариант Scale Picture - Retain Shape. По крайней мере изображение не будет искажено и разместится в отведенном ему месте. Не забудьте, что всегда можно изменить свое решение, два раза щелкнув на объекте отчета.

Выберите нужную опцию:

если изображение находится в поле типа General таблицы

если изображение находится в файле BMP или ICO



Изменение масштаба изображения
для соответствия обрисованному контуру

Изменение масштаба пропорционально изображению

Без изменения масштаба изображения

Рис. 11.13.

В окончательном виде отчет представлен на рис. 11.14.

Характеристика модели автомобиля	
Наименование модели	NAME MODEL
Фирма	NAME FIRM
Страна	NAME COUNTRY
Рабочий объем (куб см)	SWEPT VOLUME
Количество цилиндров	QUANTITY DRUM
Мощность (л с)	CAPACITY
Крутящий момент (нМ)	TORGUE
Топливо	NAME FUEL OIL
Максимальная скорость (км/ч)	TOP SPEED
Разгон до 100 км/ч (с)	STARTING
Шины	NAME TYRE

Рис. 11.14. Отчет PROPMOD в окончательном виде

Для выполнения отчета в пользовательской программе служит команда
REPORT FORM *FileName1* | ?

[Scope] [FOR *lExpression1*] [*lExpression2*]
[HEADING *cHeadingText*]
[NOCONSOLE] [NOOPTIMIZE]
[PLAIN]
[PREVIEW [NOWAIT]]
[TO PRINTER [PROMPT] | TO FILE *FileName2* [ASCII]]
[NAME *ObjectName*]
[SUMMARY]

В этой команде параметр *FileName1* определяет имя файла с описанием отчета.

Для вывода только определенных записей используется параметр *Scope*. По умолчанию выводятся все записи.

Помещаемые в отчет данные можно ограничить и с помощью опций FOR или WHILE. Данные будут печататься до тех пор, пока выражения в этих опциях будут иметь значения .T..

Для распечатки в начале каждой страницы дополнительной информации служит опция HEADING. В то же время использование опции PLAIN подавляет печать заголовков страницы (верхнего колонтитула). Таким образом, совместное применение этих двух опций обеспечивает печать дополнительного текста *cHeadingText* только на первой странице отчета.

Повлиять на содержание выводимых данных можно и с помощью опции SUMMARY, при использовании которой данные, размещенные в полосе отчета Details, выводятся не будут.

Опция NOCONSOLE позволяет предотвратить параллельный вывод данных при распечатке отчета на экран.

Опция NAME позволяет определить ссылку на объект среды окружения Data Environment. Таким образом, мы можем управлять объектами среды окружения (курсорами и отношениями) во время подготовки отчета. Если эта опция не используется, по умолчанию для ссылки на среду окружения принимается имя отчета.

Данные из отчета мы можем направить в окно предварительного просмотра, на принтер или в файл, задавая в команде опции PREVIEW, TO PRINTER или TO FILE соответственно. При выводе данных в окно предварительного просмотра можно не приостанавливать действие программы, если указать дополнительную опцию NOWAIT. При выводе данных на принтер использование дополнительной опции PROMPT обеспечивает вывод диалогового окна для уточнения установок принтера. Если мы направляем данные в файл, то по умолчанию он получает расширение TXT и в него помещаются все необходимые коды принтера для обеспечения возможности его дальнейшего вывода на соответствующее печатающее устройство. Если вы хотите получить только текст, используйте дополнительную опцию ASCII. Естественно, это приведет к потере графического оформления.

Число символов в строке будет определяться системной переменной

_ASCII_COLS = *nExpression*

Параметр *nExpression* по умолчанию равен 80.

Число строк на листе будет определяться системной переменной

`_ASCIIROWS = nExpression`

Параметр *nExpression* по умолчанию равен 63.

Таким образом, для получения текстового файла на основе данных отчета можно, например, выполнить такой код:

`_ASCII_COLS = 60`

`_ASCIIROWS = 40`

`REPORT FORM Rep_price TO FILE D:\INFO\PRICE_TEXT ASCII`

Управление режимом печати

Все или по крайней мере большинство приложений Windows позволяют для каждого документа задавать индивидуальные условия печати: размеры полей, страничную ориентацию и т. д.

Естественно предположить, что так же поступает Visual FoxPro со своими документами, предназначенными для печати, - отчетами. В команде выполнения отчета **REPORT FORM** есть опция **TO PRINTER PROMPT**, которая обеспечивает вывод на экран диалогового окна для установки условий печати. Это окно представлено на рис. 11.15.

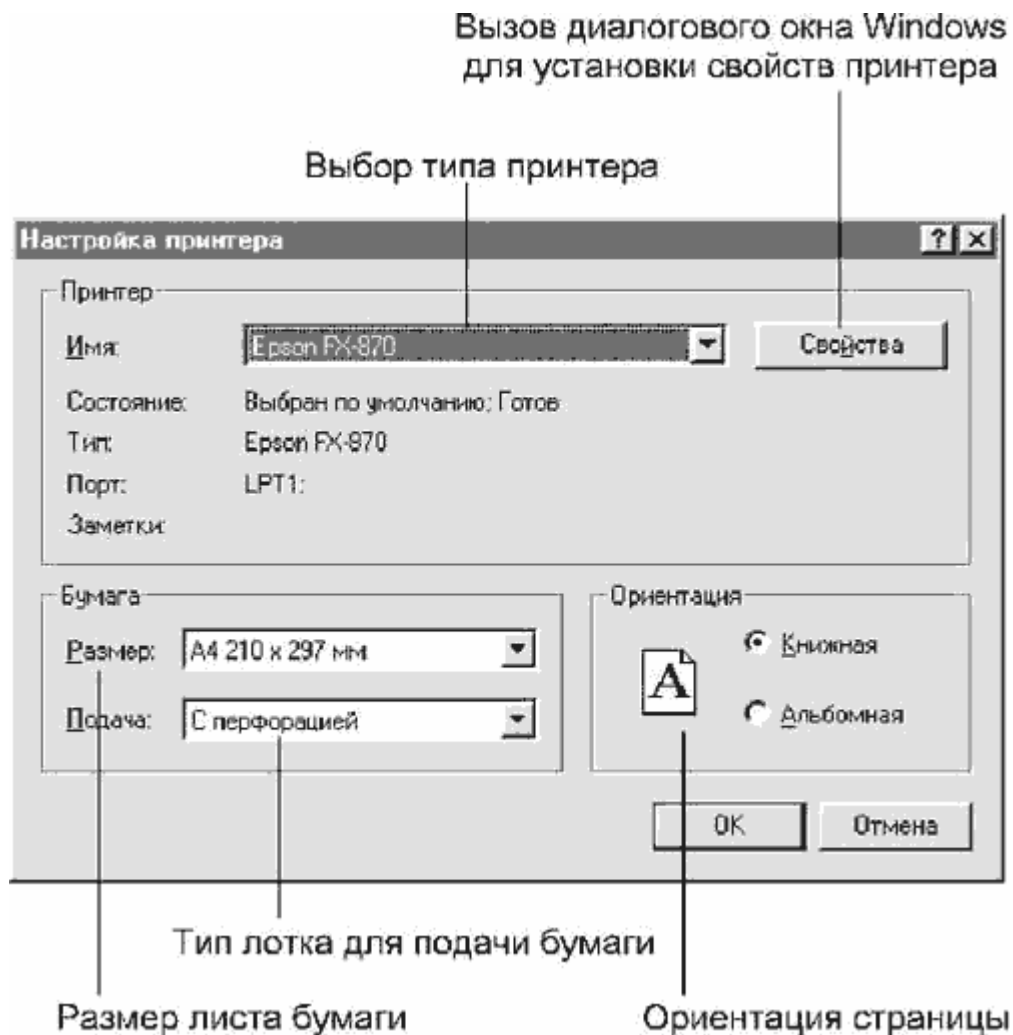


Рис. 11.15. Диалоговое окно установки принтера в Visual FoxPro

Кстати, это окно можно вывести на экран и после выполнения функции `GETPRINTER()`

Если по каким-то причинам появление этого окна в пользовательской программе нежелательно, то изменить режим печати можно и программно. Файл, в котором запоминается описание отчета, представляет собой стандартную таблицу Visual FoxPro. В первой записи этого

файла в поле `expr`, которое имеет тип поля примечаний, в текстовом виде указывается информация об установках принтера.

Каждая установка записывается с новой строки, всего в этом поле может храниться до 16 различных установок:

- **DRIVER** - имя маршрутизатора, используемого **Windows** для управления запросами на печать от клиентов. Он определяет, какой компонент спулера печати должен обрабатывать этот запрос.
- **DEVICE** - имя принтера.
- **OUTPUT** - имя порта, к которому присоединен принтер.
- **ORIENTATION** - ориентация страницы.
- **PAPERSIZE** - размер бумаги.
- **PAPERLENGTH** - длина листа бумаги.
- **PAPERWIDTH** - ширина листа бумаги.
- **SCALE** - фактор масштаба документа.
- **COPIES** - число копий, которое будет напечатано.
- **DEFAULTSOURCE** - тип лотка для подачи бумаги.
- **PRINTQUALITY** - горизонтальное разрешение принтера.
- **COLOR** - определяет, будет использоваться цветная или монохромная печать.
- **DUPLEX** - определяет, будет ли использоваться при печати двойной проход.
- **YRESOLUTION** - вертикальное разрешение принтера.
- **TTOPTION** - определяет режим печати шрифта **TrueType**.
- **COLLATE** - определяет порядок вывода страниц при печати нескольких копий.

Пример содержания поля `expr` для отчета **Propmod** приведен на рис. 11.16.

Допустимые значения для перечисленных выше параметров можно получить с помощью функции

PRINFO(*nPrinterSetting* [, *cPrinterName*])

которая возвращает текущие установки, сделанные в **Windows**.

Различные допустимые значения параметра **nPrinterSetting** приведены в следующем списке:

- 1 - ориентация страницы
- 2 - размер бумаги
- 3 - длина листа бумаги
- 4 - ширина листа бумаги
- 5 - масштаб документа
- 6 - число копий, которое будет напечатано при выводе
- 7 - тип лотка для подачи бумаги
- 8 - горизонтальное разрешение принтера
- 9 - цветная или монохромная печать
- 10 - перенасыщенная печать
- 11 - вертикальное разрешение принтера
- 12 - режим печати шрифта **TrueType**
- 13 - режим последовательной печати

Получить информацию о текущем принтере можно с помощью функции

APRINTERS(*ArrayName*)

которая записывает данные в указанный массив. Массив может до выполнения ункции не существовать. Например, команда:

APRINTERS(aCurrentPrn)

запишет в массив **aCurrentPrn** следующие данные:

ACURRENTRRN

Pub A

(1, 1) C "Epson LQ-100 ESC/P 2"

(1, 2) C "LPT1:"

Установку драйвера принтера можно проверить с помощью функции

PRINTSTATUS()

Если драйвер установлен, она возвращает значение .Т.. Обратите внимание, что эта функция не определяет готовность принтера к печати.

Как быть, если мы хотим напечатать лишь некоторые страницы из отчета? Нам поможет команда

PRINTJOB

<<Команды вывода на печать>>
ENDPRINTJOB

Она активизирует установки системных переменных, имеющих отношение к печати. Например, следующий фрагмент обеспечит вывод на печать вторую, третью и четвертые страницы отчета Propmod.

```
_PBPAGE = 2
_PEPAGE = 4
PRINTJOB
REPORT FORM Propmod TO PRINTER
ENDPRINTJOB
```

11.2. Создание отчетов в Access

В MS Access получать твердые копии результатов обработки данных можно путем распечатки таблиц, запросов и форм. Но при этом сложно или невозможно получить то качество и гибкость, которые предоставляет Конструктор отчета.

В этом параграфе мы расскажем, как создавать отчет с помощью Конструктора отчета и печатать его из программы Access.

Помимо вывода данных на печать, Конструктор отчета поможет провести, при установке соответствующих опций, предварительную сортировку и группировку данных, форматирование данных и подсчет промежуточных результатов без какого-либо программирования.

Для запуска Конструктора отчета в контейнере БД перейдите на вкладку Отчеты и нажмите кнопку Создать. В появившемся диалоговом окне в списке выберите пункт Конструктор. При загрузке Конструктора отчета вместе с ним загружается меню Конструктора отчета, а также панели инструментов для форматирования данных, выбора элементов управления и работы с Конструктором отчета.

Отчет для отображения данных из какого-либо набора должен иметь установленным свойством RecordSource (Источник данных). Обычно источник данных выбирается при первом вызове Конструктора отчета для создания конкретного отчета, когда диалог позволяет выбрать нужные таблицу или запрос из списка таблиц и запросов, находящихся в текущей базе данных. Если вам необходимо отображать данные из таблиц и запросов, находящихся в других базах данных Access или другого формата, то вам необходимо либо их присоединить, либо создать к ним сквозной запрос. Как это сделать, мы говорили в [восьмой главе](#). Если вы не указали источник данных при создании отчета, то можете исправить положение с помощью окна Properties (Свойства), которое служит для установки свойств объектов в течение дальнейших изменений в отчете.

Элементы управления можно копировать и вставлять в другие отчеты, при этом элементы переносятся из отчета в отчет со всеми установленными значениями свойств. Можно выделить и перенести целый набор объектов. Для выделения объекта вам достаточно, чтобы хотя бы часть его попала в область охвата указателя мыши, перемещаемого при нажатой левой кнопке. Для того чтобы установить эти параметры, используется диалоговое окно Параметры и вкладка Формы/Отчеты, как показано на рис. 11.17.

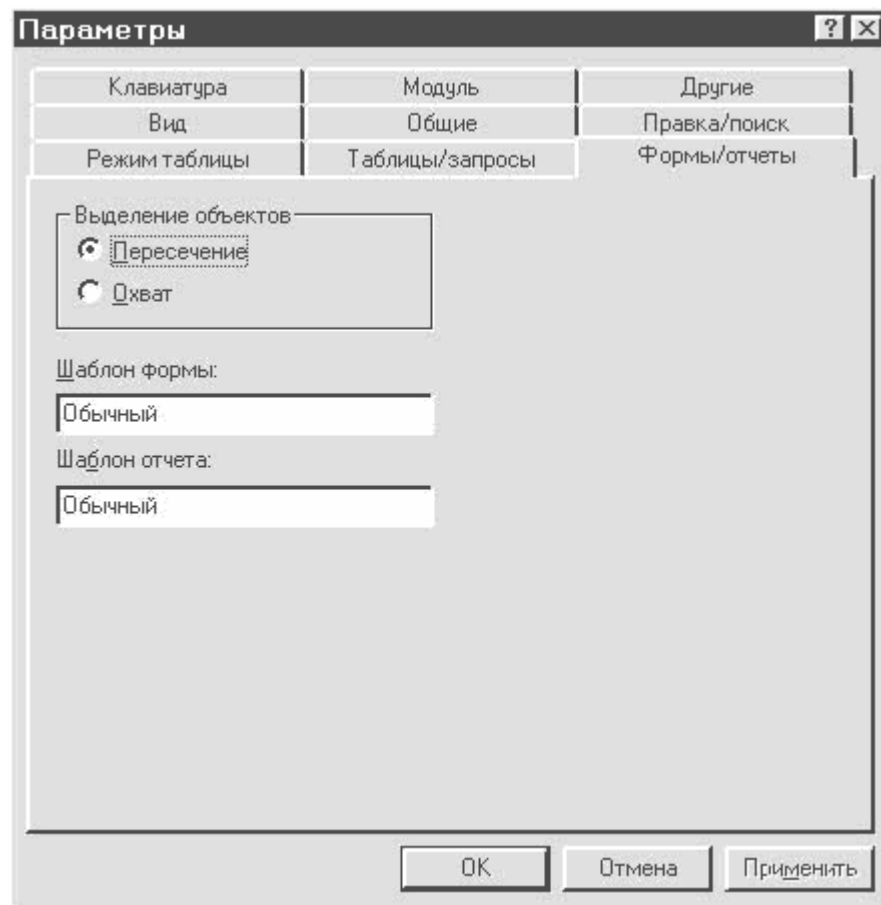


Рис. 11.17.

Если вам необходимо несколько элементов, у которых много одинаковых свойств, создайте элемент, обладающий всеми общими для группы объектов свойствами, а затем дублируйте их с помощью команды *Дублировать* меню *Правка*.

В тех случаях, когда вам необходимо особо точно выстраивать элементы в отчете, используйте линейку, которая займет место вдоль верхней и левой границ окна Конструктора отчета. Для вывода Линейки используется команда с одноименным названием в меню *Вид*. При этом шкала измерений зависит от установок Windows. Поэтому, если вдруг шкала линейки окажется в дюймах, а не в сантиметрах, обратитесь к Панели Управления Windows и выберите установку Язык и Стандарты. В ней вы найдете вкладку, в которой можете установить метрическую систему исчисления. После этого, что очень приятно, не потребуется перезагрузка Access - установки будут приняты немедленно. Вместе с линейкой можно вывести сетку, воспользовавшись командой с одноименным названием из меню *Вид*.

В любой момент создания отчета вы можете перейти в режим предварительного просмотра и обратно, то есть для того, чтобы получить представление о внешнем виде отчета, вам необязательно его печатать. Режим предварительного просмотра позволяет увидеть отчет в том виде, в котором он будет выглядеть при печати.

При работе с отчетом и его элементами часто возникает необходимость в различных операциях форматирования. Например, выравнивание группы элементов по верхнему краю, приведение их размеров к одному значению, установка одинакового расстояния между элементами в группе. Для этого используется меню *Формат*, команды *Выровнять*, *Размер*, *Интервал по горизонтали* и *Интервал по вертикали*. Если вам не нравится шрифт текстового поля и его размер, принятые по умолчанию для каждого нового объекта, вы можете создать объект с требуемыми свойствами и запомнить его в качестве стандартного. Для этого выделите созданный вами будущий стандартный объект и выберите команду *Задать стандартные свойства* в меню *Формат*.

В меню *Формат* имеется команда *Автоформат*, которая служит для быстрого выбора формата отчета из заранее predeterminedных, в которых задаются шрифт, цвет и границы. При этом вы можете с помощью кнопки *Настройка* переопределить любой встроенный формат или создать новый. Любой созданный отчет вы можете указать в качестве шаблона. Для этого используется вкладка *Формы/Отчеты* окна *Настройки параметров*.

Отчет в Access, так же как в Visual FoxPro, состоит из пяти частей: заголовок отчета, примечание отчета (итоги), верхний колонтитул, нижний колонтитул (или, другими словами, заголовок страницы и примечание страницы) и область данных. Все области, кроме области

данных, вы можете выводить или не выводить в окне Конструктора отчетов. Каждая область имеет свой набор свойств и событий.

Рассмотрим пример создания отчета, который будет выводить сведения о наличии автомобилей на складе, группируя их по странам нахождения штаб-квартир и выводя сумму товара по каждой группе. При этом каждая отдельная группа должна выводиться своим цветом шрифта и иметь изображение национального флага рядом с названием страны. Для этого нам надо создать запрос, который мы назовем "Данные для отчета" и в который отберем четыре поля из трех таблиц.

Запрос будет выглядеть следующим образом, если его открыть в режиме SQL:

```
SELECT DISTINCTROW country.country_name, model.name_model,
[automobile passenger car].date_issue, [automobile passenger car].cost
FROM country INNER JOIN (firm INNER JOIN (model INNER JOIN
[automobile passenger car] ON model.key_model = [automobile passenger
car].key_model) ON firm.key_firm = model.key_firm) ON country.key_country
= firm.key_country;
```

Начав создание отчета, выберите в качестве источника данных запрос "Данные для отчета". Попад в режим Конструктора отчета, создайте себе комфортное окружение, разместив панели инструментов так, чтобы они всегда находились в одном месте и предоставляли быстрый доступ к нужным вам кнопкам. Оставьте в панели инструментов только те элементы, которые вы собираетесь часто использовать. Поработав с Конструктором отчета буквально одну-две недели, вы поймете, что держать кнопки для вывода-удаления Линейки и Сетки в панели инструментов Конструктора отчета смысла нет, так как это довольно редкая операция, к тому же ее можно выполнить из меню. А вот кнопка Копировать формат понадобится довольно часто.

Пример размещения инструментов и окон при работе с Конструктором приведен рис. 11.18.

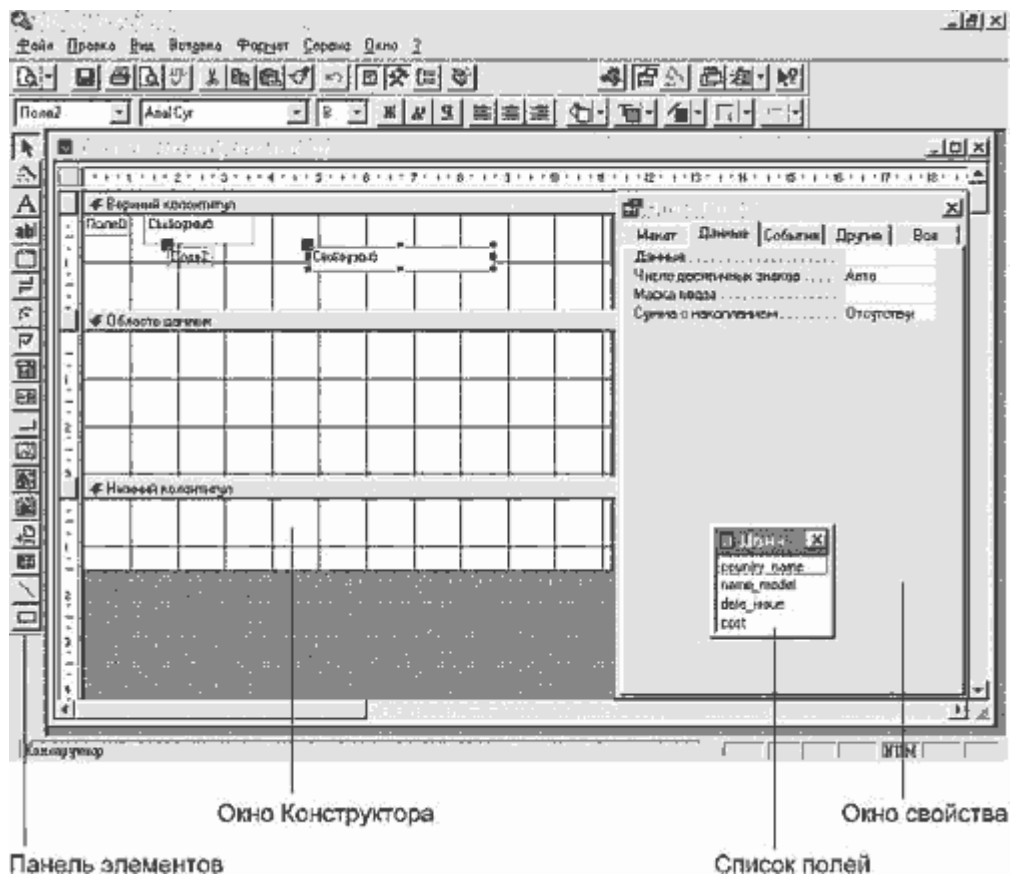
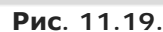


Рис. 11.18.

После того как вы разместили все окна и панели инструментов, определите, какие из частей отчета вы собираетесь использовать. Если вам не нужен заголовок отчета, то отключите его вывод.

С помощью линеек, отделяющих одну часть отчета от другой, можно изменить размер каждой из областей. Для этого необходимо установить курсор мыши на линейку и, когда он примет форму перекрестия, нажать кнопку мыши и изменить размер перетаскиванием. Для размещения



Окно Конструктора отчета изменилось, его новый вид представлен на рис. 11.20, и, надеюсь, то же самое вы наблюдаете на экранах своих компьютеров.

После того, как у нас появилась область для группирования, мы разместили в ее заголовке поле `country_name`. Теперь при выводе отчета данные, соответствующие каждой новой стране, будут предваряться ее названием.

= Sum(cost)

Обратите внимание, что выражение обязательно должно начинаться со знака равенства. В противном случае Конструктор отчета воспримет его как параметр и потребует ввести его значение перед форматированием отчета. Так как мы разместили элемент управления в Примечании группы, то нам не надо заботиться о том, сумму по каким странам он будет вычислять. Сумма будет вычислена именно по группе. Теперь имеет смысл, выбрав команду *Предварительный просмотр*, вывести на экран макет нашего отчета. В окне предварительного просмотра он должен выглядеть так, как это представлено на рис. 11.21.

Сведения по:		Германия
МВ 3.0	10 10.95	37000
ВВ 1.0	02 01.96	30000
ВВ 1.4	10 12.95	30000
ВВ 1.0	08 09.95	25000
Сумма:		122000

Сведения по:		Италия
МВ 1.3	05 10.95	10500
МВ 1.4	06 06.95	10500
МВ 1.4	07 07.94	10000
Сумма:		31000

Сведения по:		США
МВ 1.3	12 06.96	35000
Сумма:		25000

Рис. 11.21.

Перед нами осталась последняя задача - вывести флаг каждой страны в заголовке группы и динамически изменить цвет шрифта у элемента управления, выводящего название страны в заголовке группы. Добавим в заголовок элемент управления Рисунок. Нас интересует его свойство Picture.

Элементы управления отчетов не имеют свойств событий, но эти свойства присутствуют у каждой из областей отчета, в том числе и у заголовков групп. Это свойства OnFormat, OnPrint, OnRetreat.

Событие Format (Форматирование) происходит при форматировании части отчета, к которой он относится, то есть на внутреннем уровне происходит установка свойств, которые мы задали при конструировании отчета. Имеет смысл иногда сделать эти свойства динамическими, чтобы при форматировании каждой группы отдельные элементы получали разные свойства в зависимости от того, к какой группе они относятся.

Запишите следующий код для процедуры события форматирования заголовка группы:

```
Private Sub ЗаголовокГруппы0_Format(Cancel As Integer, FormatCount As Integer)
    Dim kolor As String
    kolor = country_name
    If Trim(kolor) = "Германия" Then
        Me!country_name.ForeColor = 255
        risrep.Picture = "c:\project_book\Flggerm.ico"
    End If
    If Trim(kolor) = "Италия" Then
        Me!country_name.ForeColor = RGB(0, 255, 0)
        risrep.Picture = "c:\project_book\FlgItaly.ico"
    End If
    If Trim(kolor) = "США" Then
        Me!country_name.ForeColor = RGB(0, 0, 255)
        risrep.Picture = "c:\project_book\Flgusa01.ico"
    End If
End Sub
```

Теперь при выводе нашего отчета для предварительно просмотра будет изучаться значение элемента управления country_name и после этого устанавливаться свойство ForeColor (цвет шрифта) объекта country_name и свойство Picture объекта risrep, который в нашем случае является элементом управления для вывода значка с национальным флагом страны, являющейся заголовком группы. В итоге получается отчет, который представлен на рис. 11.22.

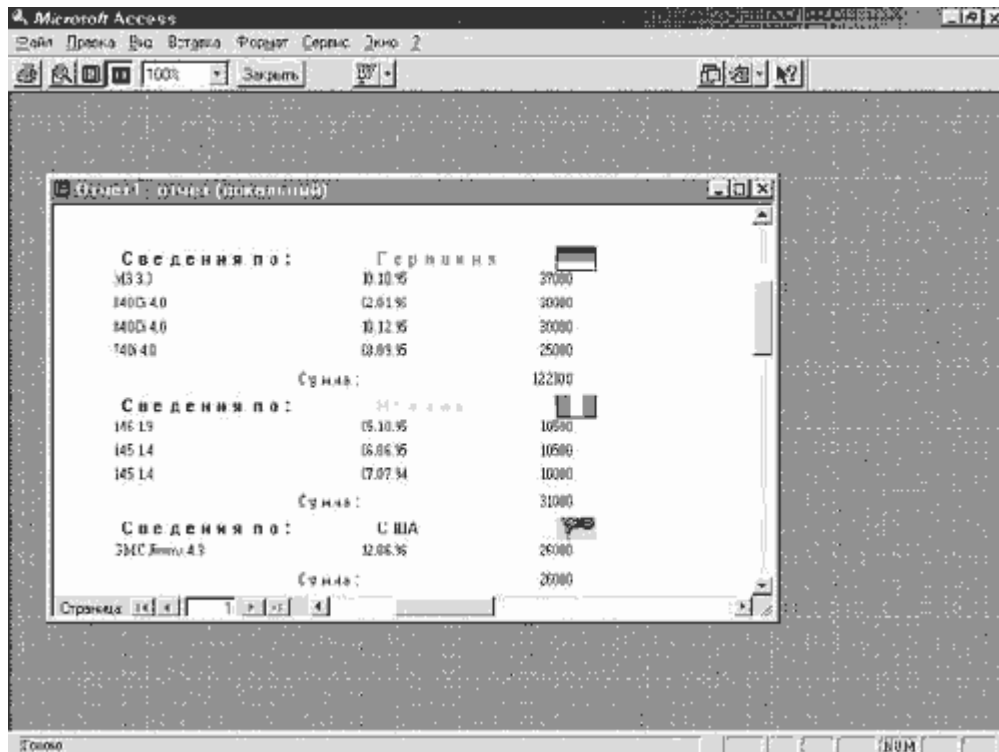


Рис. 11.22.

Использование событий придаст вашим отчетам значительную гибкость, вы сможете использовать различные итоги в зависимости от ситуации, скрывать и выводить суммы, целые колонки полей. Создав один отчет, вы фактически получите несколько его вариантов.

Кроме этого, используя событие самого отчета например, Открытие, вы сможете легко запретить нежелательному пользователю вывод вашего отчета на печать или его предварительный просмотр.

В заключение мы хотели бы дать вам следующие рекомендации при построении отчета в Access:

- максимально используйте мастера построения отчетов;
- создавайте свои автоформаты;
- используйте шаблоны, если необходимо построить большое количество схожих отчетов;
- используйте перекрестные запросы, когда попадаете в затруднительную ситуацию (например, если необходимо значения какого-нибудь поля сделать заголовками колонок);
- используйте параметрические запросы, чтобы уменьшить время предварительной обработки и сэкономить бумагу, а тем самым и кислород для себя и своих потомков;
- активно используйте свойства событий как самого отчета, так и его составных частей.

СУБД Access позволяет чрезвычайно легко организовать печать отчетов из формы. Для этого достаточно достаточно нескольких щелчков мышью.

Откройте какую-либо форму в Конструкторе формы. Выделите на панели элементов кнопку "Кнопка". Щелкните мышью на форме в месте, где должен находиться левый верхний угол будущей кнопки для печати отчета. Автоматически запустится Мастер "Создание кнопок". Выберите в списке категорий пункт "Работа с отчетом", а в списке действий - "Печать отчета". На следующем шаге из списка необходимо выбрать имя одного из созданных ранее отчетов. Далее останется выбрать значок, которым будет украшена кнопка, и назначить ей имя, например cmdPrint.

В форме появится кнопка, при нажатии на которую будет печататься требуемый отчет.

Если в Конструкторе формы вы посмотрите на код, который будет выполняться при наступлении события OnClick для этой кнопки (нажатие кнопки), то увидите следующий текст:

```
Sub cmdPrint_Click()
On Error GoTo Err_cmdPrint_Click
Dim stDocName As String
stDocName = "Account"
DoCmd.OpenReport stDocName, acNormal
Exit_cmdPrint_Click:
```

```
Exit Sub
Err_cmdPrint_Click:
    MsgBox Err.Description
    Resume Exit_cmdPrint_Click
End Sub
```

В данном примере для печати отчета используется метод **OpenReport**, который в программе выполняет макрокоманду "ОткрытьОтчет" (**OpenReport**). Этот метод имеет следующий синтаксис:

DoCmd.OpenReport *ИмяОтчета* [, *Режим*] [, *ИмяФайла*][,*Условие*]

Аргумент *ИмяОтчета* представляет имя отчета, хранящегося в текущей базе данных.

Для задания значения аргумента *Режим* может использоваться одна из следующих встроенных констант:

- **acNormal** - печать отчета (по умолчанию);
- **acDesign** - вызов Конструктора отчета;
- **acPreview** - вывод отчета в окно предварительного просмотра.

Аргумент *ИмяФайла* позволяет задать имя запроса, хранящегося в текущей базе данных, с помощью которого будут подготовлены данные для печати отчета.

Аргумент *Условие* позволяет задать допустимое предложение SQL **WHERE** без ключевого слова **WHERE**

Необходимо отметить, что в Access мы можем легко вывести на печать не только отчет, но и любой активный объект, например форму или таблицу. Для этого используется метод **PrintOut**, который выполняет макрокоманду "Печать" (**PrintOut**). Синтаксис этого метода имеет следующий вид:

DoCmd.PrintOut [*Диапазон*] [, *СоСтраницы*, *ПоСтраницу*]
[, *Качество*] [, *ЧислоКопий*] [, *РазобратьКопии*]

В качестве аргумента *Диапазон* может использоваться одна из следующих встроенных констант:

- **acPrintAll** - для печати всего отчета (по умолчанию);
- **acSelection** - для печати фрагмента отчета;
- **acPages** - для печати указанных страниц. При этом должны быть указаны аргументы *СоСтраницы* и *ПоСтраницу*.

Для задания значения аргумента *Качество* должна использоваться одна из следующих встроенных констант:

- **acHigh** - высокое качество печати (разрешение принтера) - по умолчанию;
- **acMedium** - среднее качество печати;
- **acLow** - низкое качество печати;
- **acDraft** - печать черновика.

Число печатаемых копий отчета задается аргументом *ЧислоКопий* которое по умолчанию равно 1.

Задание для аргумента *РазобратьКопии* значения, равного **True(-1)**, определяет печать с раскладкой по копиям, а **False (0)** - печать без раскладки. Если оставить данный аргумент пустым, будет принято значение по умолчанию (**True**).

Напомним, что в Access необязательный аргумент посреди списка аргументов разрешается пропустить, однако при этом необходимо ввести запятую, отделяющую пропущенный аргумент. Если опускаются один или несколько последних аргументов, вводить запятые вслед за последним указанным аргументом не требуется.

Например, для вывода формы на принтер мы можем создать в ней специальную кнопку, для события **OnClick** которой запишем следующий код, при выполнении которого будут напечатаны три экземпляра отчета:

```
Private Sub cmdForm_Click()
DoCmd.PrintOut(acPrintAll,,3)
```

End Sub

Глава 12

Подготовка и отладка пользовательского приложения

- 12.1. Общие принципы отладки приложения
- 12.2. Инструментальные средства отладки
 - Отладка программы в Visual FoxPro
 - Отладка программы в Access
 - Обработка ошибок процессора баз данных в Access
 - Отладка программы в Visual Basic
- 12.3. Подготовка приложения для распространения

12.1. Общие принципы отладки приложения

При разработке программ даже опытным специалистам не удастся избежать ошибок, и в этом случае при запуске программа работает не так, как задумывалось, или не работает вовсе.

В этом параграфе мы классифицируем ошибки, которые могут возникнуть при разработке пользовательского приложения, и опишем общие принципы отладки программы, реализуемые в рассматриваемых средствах разработки.

Ошибки программы можно разделить на три группы:

- Синтаксические ошибки связаны, как правило, с неправильным написанием команд и функций.
- Ошибки выполнения программы происходят чаще всего из-за отсутствия вызываемых компонентов, например открываемых таблиц, вызываемых функций, объектов и т. д.
- Логические ошибки приводят к неправильным результатам, несмотря на безукоризненную работу программы. Их причиной чаще всего является использование не соответствующих алгоритму данных или выполнение вычислений в неправильной последовательности. Это самые трудные для выявления ошибки.

Для выявления ошибок в пользовательской программе разработчик может использовать в Visual FoxPro следующие средства:

- Окно *Trace* для отслеживания процесса выполнения программы.
- Окно *Debug* для наблюдения за значениями переменных, полей, возвращаемых значений функций и задания условий приостановки выполнения программы.
- Окно *Command*, позволяющее в большинстве случаев во время ожидания программой действий пользователя выполнить интересующие действия или быстро вывести текущие значения.
- Можно добавить в программу специальный отладочный код, который будет выводить контрольные сообщения и другую важную для отладки информацию. В этом случае чаще всего используются команды **WAIT WINDOW**, **?** или функция **MESSAGEBOX()**. Можно также периодически сохранять значения переменных с помощью команды **LIST MEMORY TO FILE**.

Access и Visual Basic имеют менее богатый набор средств отладки. В то же время проверка синтаксиса набираемого программного кода выполнена в них более удобно. При написании программы в Visual Basic, а следовательно, и в Access синтаксические ошибки перехватываются уже при переходе на следующую строку при редактировании кода процедуры или функции или сразу перед исполнением кода. Например, если при создании конструкции выбора с помощью оператора **If** вы, написав выражение условия, не завершаете ее оператором **Then**, то немедленно получите сообщение об ошибке, и строка с неправильной командой будет выделена:

```
Sub Form_After_Update()
  If me![txtTeam]>> "Boston"
    ...
```

End If
End Sub

Простые ошибки, связанные с неправильным написанием команд, в Visual FoxPro выявляются при компиляции программы. Сообщения о таких ошибках выводятся на экран и сохраняются на диске в файле с именем, одинаковым с именем программы и имеющим расширение ERR, если сделана установка

SET LOGERRORS ON | off

которая сохраняет все сообщения об ошибках в текстовом файле.

SET DEVELOPMENT ON | off

проверяет дату и время создания программного файла и, если файл с исходным кодом создан позднее, чем с объектным (после компиляции), автоматически перекомпилирует его перед выполнением.

Эти установки можно сделать и в диалоговом режиме, выбрав вкладку General в окне Options, вызвать которое можно в меню *Tools*.

12.2. Инструментальные средства отладки

В этом параграфе мы рассмотрим основные методы и способы отладки пользовательского приложения в Visual FoxPro, Access и Visual Basic.

Отладка программы в Visual FoxPro

Для выявления ошибок, связанных с неправильными вычислениями или ветвлением программы, в FoxPro имеется целый комплекс специальных средств, делающих работу по отладке программы быстрой и эффективной. При проверке сомнительных мест в программе поможет команда *Trace Window* из меню *Tools* или установка

SET ECHO on | OFF

которая активизирует окно трассировки *Trace* для отслеживания работы программы по ее исходному тексту, так как текущая выполняемая команда в этом окне выделяется. На рис. 12.1 приведено окно *Trace* и даны пояснения для его основных элементов. С помощью команд, расположенных в меню окна, можно выполнить следующие действия:

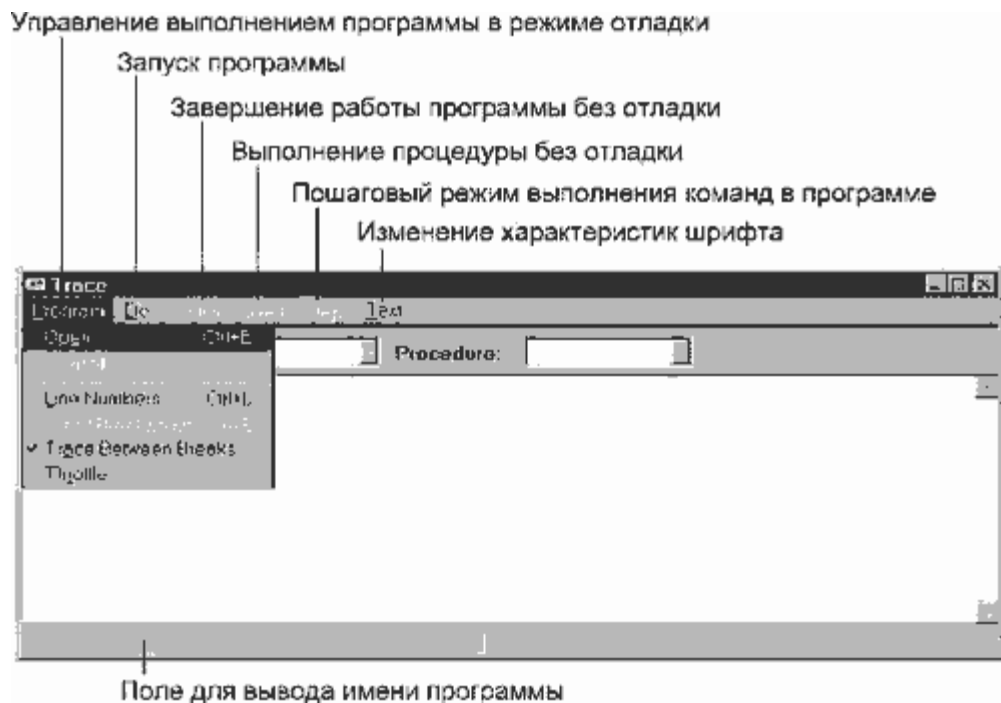


Рис. 12.1.

- **Open** - открытие программы.
- **Cancel** - прерывание работы программы.
- **Line Numbers** - нумерация программных строк.
- **Clear Breakpoints** - удаление точек останова.
- **Trace Between Breaks** - управление режимом просмотра команд в окне *Trace*.
- **Throttle** - установка задержки времени выполнения команд в программе.

Для наблюдения за выполнением программы с помощью окна *Trace* необходимы следующие действия:

1. Выполните команду *Trace Window* из меню *Tools*.
2. Откройте требуемую программу с помощью команды *Open* меню *Program* окна *Trace*.
3. Установите задержку выполнения кода для возможности визуального наблюдения с помощью команды *Throttle* этого же меню. Время задержки придется подобрать опытным путем, так как этот показатель зависит от сложности программы и быстродействия компьютера.
4. Запустите программу на выполнение, выбрав меню *Do* окна *Trace*.

Если вам необходимо тщательное наблюдение за результатом выполнения каждой строки кода, вы можете выбрать на четвертом этапе пошаговое выполнение программы, щелкнув на меню *Step!* окна *Trace*. В этом случае следующая строка программы будет выполняться только после очередного щелчка на этом меню.

В этом режиме выбор меню *Out!* позволяет отменить пошаговый режим и продолжить выполнение программы до первой строки программы более высокого уровня, следующей за командой, вызвавшей текущую программу. Если программы более высокого уровня нет, завершится выполнение текущей программы.

Прервать выполнение программы можно нажатием клавиши **Esc**. Если необходимо приостановить выполнение программы в определенных точках, щелкните мышкой на поле слева от нужной строки. Слева от строки появится отметка в виде точки. Каждый раз при достижении этого места выполнения программы будет приостанавливаться, что позволит просмотреть содержимое таблиц, значений всех временных переменных и выполнить другие действия, связанные с контролем состояния среды, а затем продолжить (**Resume**) или прервать работу программы (**Cancel**). Отменить приостановку можно повторением указанных действий.

Расширить возможности трассировки позволяет установка

SET DEBUG ON | off

которая дает возможность контролировать значения любых выражений и задавать условия приостановки выполнения программы во время ее работы (команда *Debug Windows* в меню *Tools*). Наберите идентификаторы временных переменных, элементов массива, стандартных функций или полей таблиц в левой части окна отладки, после набора каждого идентификатора нажимайте клавишу **Enter**. При выполнении программы значения этих выражений будут выводиться в правой части окна.

В окне отладки можно устанавливать контрольные точки для приостановки работы программы при смене значений, указанных в левой части выражений. Нажмите клавишу пробела или кнопку мыши напротив нужного выражения в колонке, которая разделяет левую и правую части окна: появится отметка в виде точки, снять отметку можно, повторив указанные действия.

Если вы хотите просматривать написанный вами код событий и методов при работе формы, то нам потребуется использовать окна *Trace* и *Debug* совместно, как это показано на рис. 12.2. Для этого выполните следующие действия:

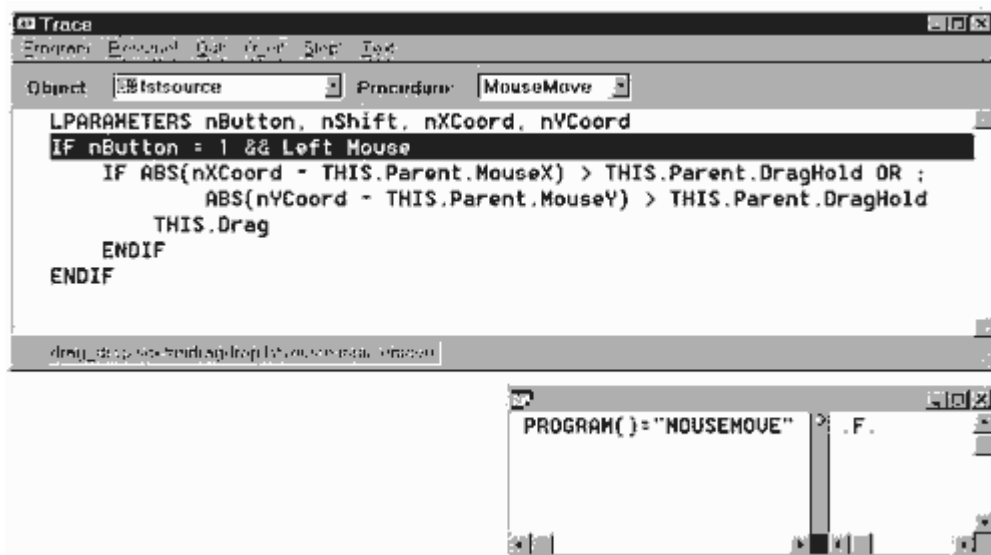


Рис. 12.2.

1. Откройте окно *Debug*.
2. Наберите в его левой части строку типа **PROGRAM()="MOUSEMOVE"**, в которой необходимо указать имя события или метода, код которого вас интересует. Установите на разделительной линейке точку останова.
3. Откройте окно *Trace*.
4. С помощью меню *Do* окна *Trace* запустите нужную форму.
5. При выполнении события *MouseMove* окно *Trace* будет активизироваться и вы сможете просматривать код или задавать его пошаговое выполнение.

Эффективность отладки в Visual FoxPro, особенно в период опытной эксплуатации программы, могут повысить следующие команды.

SET ALTERNATE TO [FileName [ADDITIVE]]

Создает на диске текстовый файл с именем *FileName* (по умолчанию расширение TXT), в котором дублируются результаты выполнения всех команд, кроме полноэкранных, что позволяет контролировать процесс работы программы даже без присутствия разработчика (в период опытной эксплуатации) и определять причины сбоя или неправильных действий оператора. Конечно, эта процедура требует достаточного дискового пространства и несколько замедляет работу программы. Без параметров команда закрывает текущий альтернативный файл. С опцией **ADDITIVE** добавляет данные в файл, в противном случае его содержимое перезаписывается.

SET ALTERNATE on | OFF

Включает (ON) или выключает (OFF) режим записи данных в альтернативный файл.

ON ERROR [Command]

Устанавливает режим ожидания для ошибочных ситуаций во время работы программы. В случае возникновения ошибки выполняется указанная команда. Как правило, это команда запуска специальной процедуры обработки ошибок.

Без аргумента команда восстанавливает режим обработки ошибочных ситуаций Visual FoxPro. При возникновении ошибки Visual FoxPro приостанавливает работу программы и при наличии исходного кода выводит на экран окно *Trace*, в котором строка кода, вызвавшего ошибку, выделена. Очевидно, что даже в отлично отлаженной программе могут возникать ошибочные ситуации хотя бы по внешним причинам, связанным со случайным удалением файлов и т. д. Поэтому любая пользовательская программа должна иметь специальный блок обработки ошибок, основу которого и будет составлять команда **ON ERROR**. Пример такой программы мы приведем в конце этого параграфа, а логика, которая должна быть заложена в блок обработки ошибок, демонстрируется на рис. 12.3.



Рис. 12.3.

В Visual FoxPro мы можем расширить возможности обработки ошибок за счет использования локальной идентификации каких-либо специфических ошибок, возникающих при выполнении тех или иных действий. Например, в форме мы можем обработать ошибочные ситуации, возникающие при выполнении каких-либо методов, без обращения к установке **ON ERROR**, то есть внутри формы. Для этого можно использовать событие **Error**.

PROCEDURE *Object.Error*
LPARAMETERS [*nIndex*,] *nError*, *cMethod*, *nLine*

Параметр *nIndex* позволяет сослаться на элемент управления по его номеру в массиве элементов управления объекта-контейнера (формы). Параметр *nError* содержит номер ошибки Visual FoxPro, *cMethod* - имя метода, который вызвал ошибку, *nLine* - номер строки внутри метода или определяемой пользователем функции, которая вызвала ошибку.

В то же время, если при обработке ошибки в процедуре обработки события **Error** возникнет еще одна ошибка, Visual FoxPro вызовет обработчик, указанный в установке **ON ERROR**, или, если такой обработчик не указан, приостановит выполнение программы.

Например, мы можем в событие **Error** формы записать код, который будет выполняться, если невозможно открыть связанные с формой таблицы, так как администратор БД проводит переиндексацию файлов:

```

LPARAMETERS nError, cMethod, nLine
* Ошибка "File in use by another"
IF nError = 108
  =MESSAGEBOX("Данные не доступны! " + ;
    "Попробуйте открыть форму позднее.")
ELSE
  * В случае другой ошибки вызываем общую процедуру
  * обработки ошибок, установленную в главной программе
DO proc_error
ENDIF
  
```

Если вы определяете локальную процедуру обработки ошибок для класса, то при использовании в пользовательском приложении объектов, основанных на этом классе, заданная процедура будет автоматически в нем задействована. Причем эта процедура обработки ошибок будет автоматически наследоваться в подклассах, основанных на этом классе. Следует иметь в виду, что если вы используете в форме какие-либо элементы управления, для которых в событии **Error** обработка ошибок не предусмотрена, то событие **Error** для формы вызвано не будет. В этом плане использование установки **ON ERROR** более универсально, так как она будет обработана не зависимо от места возникновения ошибки.

Проверить реакцию программы обработки ошибок можно, искусственно создав ошибочную ситуацию с помощью команды

ERROR *nErrorNumber* | *nErrorNumber*, *cMessageText1*
[*cMessageText2*]

которая генерирует ошибку Visual FoxPro. Параметр *nErrorNumber* определяет номер ошибки, который используется для воспроизведения стандартного сообщения Visual FoxPro. Параметр *cMessageText1* задает текст, появляющийся в сообщении об ошибке, который может содержать дополнительную информацию. Например, если вы ссылаетесь на переменную, которой не существует, Visual FoxPro может вывести имя этой переменной в сообщении об ошибке. Параметр *cMessageText2* определяет текст, отображаемый в сообщении об ошибке. Когда вместо параметра *nErrorNumber* в команде задается параметр *cMessageText2*, будет сгенерирована ошибка Visual FoxPro с номером 1098 (определяемая пользователем ошибка). Чтобы переместить часть сообщения об ошибке на следующую строку, используйте в *cMessageText2* символ возврата каретки CHR(13).

Команда **ERROR** может использоваться помимо проверки программы обработки ошибок для того, чтобы показать особые сообщения об ошибках. Если действует установка **ON ERROR**, то при выполнении команды **ERROR** Visual FoxPro выполняет процедуру обработки ошибок, определяемую в установке **ON ERROR**. Если происходит ошибка для объекта, будет выполнено событие Error для этого объекта.

Если вы задаете команду **ERROR** из окна *Command* и установка **ON ERROR** не действует, Visual FoxPro показывает сообщение об ошибке. Если команда **ERROR** выдана в программе и установка **ON ERROR** также не действует, Visual FoxPro выводит сообщение об ошибке и разрешает вам закончить или приостановить программу, либо игнорировать ошибку.

Например, чтобы задать появление ошибки Visual FoxPro номер 12, напишем следующую команду в требуемом месте программы:

ERROR 12

Появится сообщение об ошибке "Variable not found" ("Переменная не найдена"). Если задать эту команду в следующем виде:

ERROR 12, `nVar1'

появится сообщение об ошибке "Variable `nVar1' not found" ("Переменная `nVar1' не найдена").

Для получения более обширной информации при отладке программы можно использовать следующие функции.

AERROR(ArrayName)

Создает массив, содержащий информацию относительно самой последней ошибки Visual FoxPro, OLE или ODBC. Параметр *ArrayName* определяет имя создаваемого массива. Функция **AERROR()** создает массив с шестью столбцами и возвращает число строк в массиве, которое определяется типом ошибки. Следующий список описывает содержание каждого элемента массива:

Номер элемента	Описание
1	Числовой. Содержит номер ошибки. Идентичен значению, возвращаемому функцией ERROR() .
2	Символьный. Текст сообщения об ошибке. Идентичен значению, возвращаемому функцией MESSAGE() .
3	Значение NULL. Если ошибка имеет дополнительный параметр, содержит текст параметра ошибки. Идентичен значению, возвращаемому функцией SYS(2018) .
4	Значение NULL. Однако, как и в

предыдущем случае, может содержать номер рабочей области, в которой произошла ошибка.

- | | |
|---|--|
| 5 | Значение NULL . Если триггер не смог выполниться (ошибка 1539), будет содержать одно из следующих числовых значений:
1 - триггер вставки потерпел неудачу;
2 - триггер модернизации потерпел неудачу;
3 - триггер удаления потерпел неудачу. |
| 6 | Значение NULL |
| 7 | Значение NULL . |

В следующем списке описано содержимое каждого элемента создаваемого массива, когда происходят ошибки OLE с номерами 1427 или 1429:

Номер элемента	Описание
1	Числовой. Содержит код ошибки 1427 или 1429.
2	Символьный. Текст FoxPro сообщения об ошибке.
3	Символьный. Текст OLE сообщения об ошибке.
4	Символьный. Имя прикладной программы (например, Microsoft Excel).
5	Значение NULL или Символьный. В последнем случае содержит имя справочного файла, где может быть найдена более подробная информация относительно ошибки.
6	Значение NULL или Символьный. В последнем случае содержит идентификатор для соответствующей темы оперативной подсказки, если информация доступна из приложения.
7	Числовой. Номер ошибки OLE.

Ниже приводится описание содержимого каждого элемента массива, когда происходит ошибка ODBC с номером 1526 (ошибка соединения ODBC). Массив может содержать две или больше строки - по одной для каждой ошибки ODBC:

Номер элемента	Описание
1	Числовой. Содержит номер ошибки 1526.
2	Символьный. Текст сообщения об ошибке.
3	Символьный. Текст ODBC сообщения об ошибке.
4	Символьный. Текущее состояние ODBC SQL.
5	Числовой. Номер ошибки ODBC источника данных.
6	Числовой. Указатель соединения ODBC.
7	Значение NULL .

ERROR()

Возвращает номер ошибки, которая произошла при выполнении программы, если действует команда **ON ERROR**.

LINENO([1])

Возвращает номер строки программы относительно первой строки главной программы, в которой произошла ошибка. Если вы указываете параметр 1, то номер строки вычисляется относительно первой строки текущей программы или процедуры.

Счетчик строк в программе учитывает все строки, включая комментарии и пустые строки. Вы всегда можете задать приостановку выполнения программы, если откроете окно *Debug* на время выполнения программы и в его левой части запишете, например, **LINENO(1) = 100**. После этого не забудьте на разделительной линии окна *Debug* поставить точку останова.

MESSAGE([1])

Возвращает сообщение об ошибке, с помощью опции 1 можно вывести последнюю программную строку, при выполнении которой произошла ошибка. Для работы функции необходимо установить **ON ERROR**.

PROGRAM([nLevel])

Возвращает название программы, во время работы которой произошла ошибка. Численный аргумент определяет уровень вложенности программ, чьи имена необходимо запомнить, если $nLevel = 1$, то возвращается имя только первой исполнявшейся программы (главной), если аргумент не указан, возвращается имя текущей программы.

Работу описанных выше функций можно проиллюстрировать следующим простым примером:

```
ON ERROR DO proc_error WITH;
ERROR(),MESSAGE(),MESSAGE(1),PROGRAM(),LINENO()
...
RETURN
PROCEDURE proc_error
LPARAMETER nError, cMess, cMess1, cProg, nLineno
? "Номер ошибки: " + LTRIM(STR(nError))
? "Сообщение об ошибке: " + cMess
? "Текст строки, вызвавшей ошибку: " + cMess1
? "Номер строки с ошибкой: " + LTRIM(STR(nLineno))
? "Ошибка произошла в программе: " + cProg
RETURN
```

Отладка программы в Access

Четыре из наиболее часто используемых функций отладки доступны через панель инструментов Access. Их описание приведено в табл. 12.1. На рис. 12.4 представлена панель инструментов для выполнения отладки и приведено описание ее кнопок.

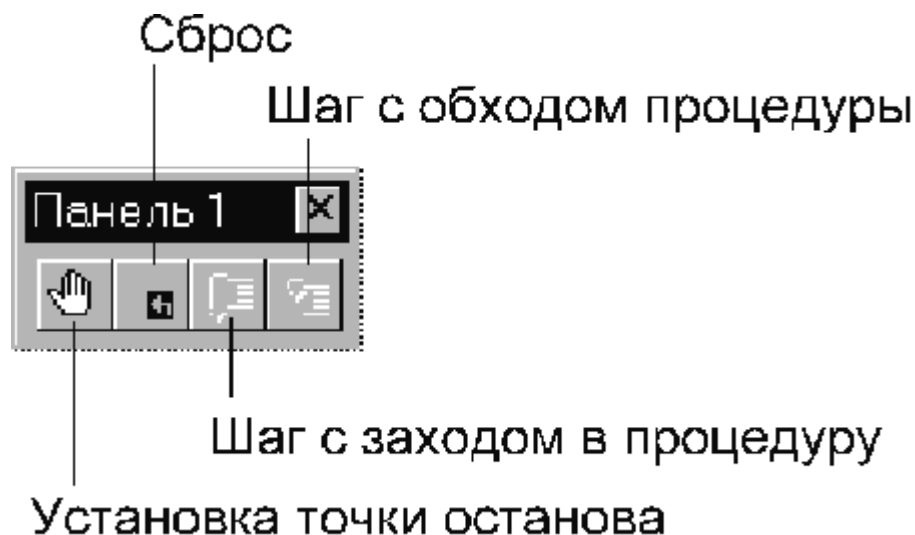


Рис. 12.4. Наиболее часто используемые функции отладки в Access

Таблица 12.1. Инструментальные функции отладки в Access

Кнопка отладки	Выполняемое действие	Соответствующая горячая клавиша
----------------	----------------------	---------------------------------

Установка точки останова	Создает или уничтожает точку останова. Точка останова	F9
Шаг с заходом в процедуру	В окне просмотра кода при вызове другой процедуры вы видите ее код	F8
Шаг без захода в процедуру	В окне просмотра кода при вызове другой процедуры вы не видите ее кода	Shift+F8
Сброс	Прекращает работу процедуры	-

Состояние прерывания приостанавливает работу программы и дает вам картину состояния на текущий момент исполнения программы.

СУБД Access попадает в состояние прерывания, когда происходит одно из следующих событий:

- Программа достигает строчки с точкой останова.
- Исполняемый код достигает строчки с выражением **Stop**.
- Выражение в строке кода генерирует неперехватываемую ошибку.

Как только **Access** встречается с одним из вышеприведенных условий, исполняемый код приостанавливается. Появляется окно отладки. При этом вы можете воспользоваться панелью инструментов Access, но не можете переключиться в другие части вашего приложения.

В состоянии прерывания значения переменных и свойств сохраняются, таким образом, вы можете:

- просматривать значения переменных, свойств и выражений;
- изменять значения переменных и свойств.

В окне отладки вы можете просматривать значения выражений и переменных в процессе перемещения между выражениями в вашем коде. Помимо этого вы можете использовать окно отладки для изменения значений переменных и свойств, чтобы исследовать, как новые значения повлияют на работу вашего кода.

Окно отладки выводит информацию, которая является результатом выполнения кода либо выражений, которые вы набираете непосредственно в окне. Например, при отладке или экспериментах с кодом у вас может появиться желание протестировать процедуру, получить значения выражений или присвоить новые значения переменным и свойствам. Все это можно сделать в окне отладки. При этом надо помнить о том, что видимость переменных, переназначенных в окне отладки, ограничена только текущей процедурой.

Вывод данных в окне отладки можно легко осуществить с помощью символа вопросительного знака, который здесь вполне успешно заменяет метод Print (рис. 12.5). Вопросительный знак означает совершенно то же самое, что и Print, и может использоваться в любом контексте, в котором может использоваться метод Print.



Рис. 12.5.

Внутри вашего кода вы можете использовать метод `Print` объекта `Debug`, для того чтобы послать вывод в окно отладки. Этот способ позволяет вам создать историю значений переменной или свойства в процессе выполнения в окне отладки. Когда приложение приостановится или закончит свою работу, вы сможете увидеть напечатанные значения. Например, следующее значение выведет в окно отладки значение переменной `myvar`:

```
Debug.Print "Значение переменной равно =" & myvar
```

Эта техника хорошо помогает, когда имеется место в программе, где переменная (в нашем случае `myvar`) изменяется. Например, вы можете использовать выражение, которое меняет значение переменной `myvar`, в цикле.

Вы можете работать с диалоговым окном **Вызовы (Calls)**, приведенном на рис. 12.6, для исследования последовательности действий приложения в случаях, если оно состоит из цепочки вложенных вызовов процедур.

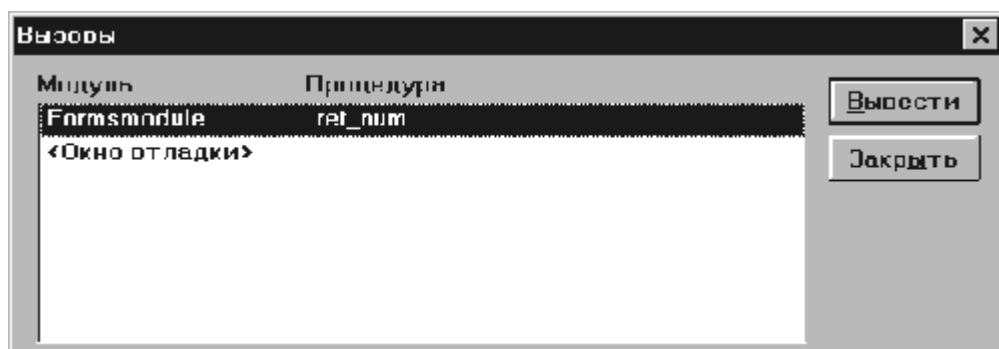


Рис. 12.6. Диалоговое окно Вызовы

Для того чтобы вызвать диалоговое окно **Вызовы**, необходимо:

- Приостановить выполнения кода. Вы можете вывести диалоговое окно **Вызовы**, только когда выполнение кода приостановлено.
- Щелкнуть на кнопке **Вызовы** в панели инструментов.

Например, процедура события может вызвать вторую процедуру, которая, в свою очередь, вызывает третью - и все это до того, как процедура события, вызвавшая эту цепочку, завершилась.

Диалоговое окно **Вызовы** выводит список всех активных процедур в серии вложенных вызовов процедур. Чем позже процедура вызывалась, тем ближе к верху диалогового окна она расположена. В информацию о каждой процедуре входит название модуля, включающего процедуру, за которым следует имя вызываемой процедуры.

Одна из главных задач, которая стоит перед программистом, создающим профессиональное приложение, - мягкий перехват ошибки, без завершения работы программы и вывода малопонятного сообщения об ошибке.

Источник ошибок времени выполнения не всегда очевиден, часто требуется потратить много часов на отладку, для того чтобы найти их. Например, хотя приложение будет верно работать при нормальных условиях, оно может вывести сообщение об ошибке, если будет введено значение не того типа, которое требует какое-либо из полей ввода.

Ошибки времени выполнения происходят тогда, когда Access получает команду, которую он не может выполнить, такую, как, например, деление на ноль. Когда Access наталкивается на ошибку времени выполнения, программа останавливается и выводится диалоговое окно с сообщением, описывающим ошибку.

В это время вы можете:

- Остановить выполнение программы.
- Продолжить выполнение, если вы разрешите проблемы в окне отладки.
- Просмотреть окно отладки, где в это время вы можете увидеть выделенной строчку, которая привела к ошибке.
- Перейти прямо к строчке, которая вызвала ошибку.
- Вызвать Справку по данной ошибке.

Для чего перехватывать ошибки времени выполнения? Перехватывая ошибки времени выполнения, вы делаете приложение более устойчивыми по отношению к типичным ошибкам, с которыми оно может встретиться.

При этом:

- Создаются устойчивые приложения. Приложение, которое перехватывает ошибки времени выполнения, может справиться со многими пользовательскими ошибками без остановки приложения. Сделав приложение устойчивым по отношению к часто встречающимся ошибкам, вы уменьшаете вероятность его полного развала в процессе работы.
- Если приложению все же не удастся обработать сложившуюся ошибочную ситуацию, ваше приложение может тем не менее аккуратно закрыть все открытые файлы и тем самым с большой долей вероятности сохранить их для дальнейшего использования.

Большая часть обработчиков ошибок имеет общую схему. Когда Access сталкивается с ошибкой времени выполнения, он ищет выражение **On Error Goto <<выражение>>**. Если он его находит, ошибка обрабатывается и выполнение продолжается либо с того же выражения, на котором оно остановилось, либо с какого-нибудь другого.

Если Access не может найти выражение **On Error Goto**, выполнение процедуры приостанавливается и Access выводит сообщение об ошибке времени выполнения, которое может в какой-то степени удивить ваших пользователей.

Существует восемь стандартных функций и выражений для перехвата ошибок. Их описание приведено в табл. 12.2.

Таблица 12.2. Основные программные функции отладки в Access

Выражение или функция	Описание
On Error Goto	Подключает обработчик ошибок и указывает расположение обработчика внутри процедуры. Может также использоваться для отключения обработчика ошибок
Err (Выражение)	Устанавливает свойство Err в указанное значение
Err (Функция)	Возвращает статус ошибки
Error (Выражение)	Воспроизводит ошибку
Error (Функция)	Возвращает сообщение об ошибке, соответствующее указанному номеру
Resume	Продолжает выполнение с выражения, которое вызвало ошибку, после того как обработчик ошибок

	отработал
Resume Next	Продолжает выполнение со строки, следующей за строкой, которая вызвала ошибку, после того как обработчик ошибок отработал
Resume Строка	Продолжает выполнение с указанной строки и метки после обработки ошибки

Существуют три действия, которые можно применить к большинству обработчиков ошибок:

- Установка перехватчика ошибок. Каждая процедура или функция, которая поддерживает перехват ошибок, должна включать выражение **On Error**, которое указывает **Access**, где искать инструкции по обработке ошибок. Хотя команда **On Error** должна указывать на метку или строку внутри той же самой процедуры, выражение после метки может вызывать другую процедуру.
- Оформление обработчика ошибок. Обработчик ошибок обычно состоит из выражения **Select Case** (или похожего выражения для принятия решений), которое различает разные значения свойства **Err** и способ их обработки.
- Выход из обработчика. Используйте одно из выражений **Resume** для указания выхода из процедуры, если ошибка приводит к тупиковой ситуации, или для продолжения выполнения программы.

Для отключения обработчика ошибок вставьте следующее выражение в код вашей процедуры:

On Error Goto 0

Когда **Access** наталкивается на ошибку времени выполнения, он ищет неактивный обработчик ошибок в следующей последовательности:

1. Текущая процедура.
2. Процедуры, перечисленные в списке Вызовы (начиная с самой последней вызываемой процедуры).
3. Обработчик ошибок внутри **Access**, который останавливает выполнение программы и выводит диалоговое окно ошибки времени выполнения.

Обработка ошибок процессора баз данных в Access

Ошибки, связанные с работой процессора баз данных, вызывают событие **Error**. Это относится к ошибкам процессора баз данных **Microsoft Jet**, но не к ошибкам времени выполнения **Access**. Выполняя процедуру события или макроса при возникновении события **Error**, вы можете предотвратить вывод сообщения об ошибке **Microsoft Access** и вывести пользовательское сообщение об ошибке, которое может оказаться более подходящим по контексту для вашего приложения.

Свойство **OnError** используется для того, чтобы указать макрос или процедуру **Access**, которая будет выполняться при возникновении события **Error**. Устанавливайте это свойство тем же путем, которым вы устанавливаете остальные свойства.

Например, синтаксис для процедуры события **Error** формы должен иметь следующую конструкцию:

```
Sub Form_Error(DataErr As Integer, Response As Integer)
```

Аргумент **DataErr** является кодом ошибки, возвращаемым функцией **Err** при возникновении ошибки. Вы можете использовать аргумент **DataErr** вместе с функцией **Error\$**, чтобы соотнести номер ошибки с соответствующим сообщением об ошибке. Аргумент **Response** определяет, будет ли выводиться сообщение об ошибке. Для задания значения этого аргумента можно использовать одну из следующих констант:

- **DATA_ERRCONTINUE** - это значение приводит к игнорированию ошибки, и код продолжается без вывода сообщения об ошибке. Вы можете добавить свое собственное сообщение об ошибке.

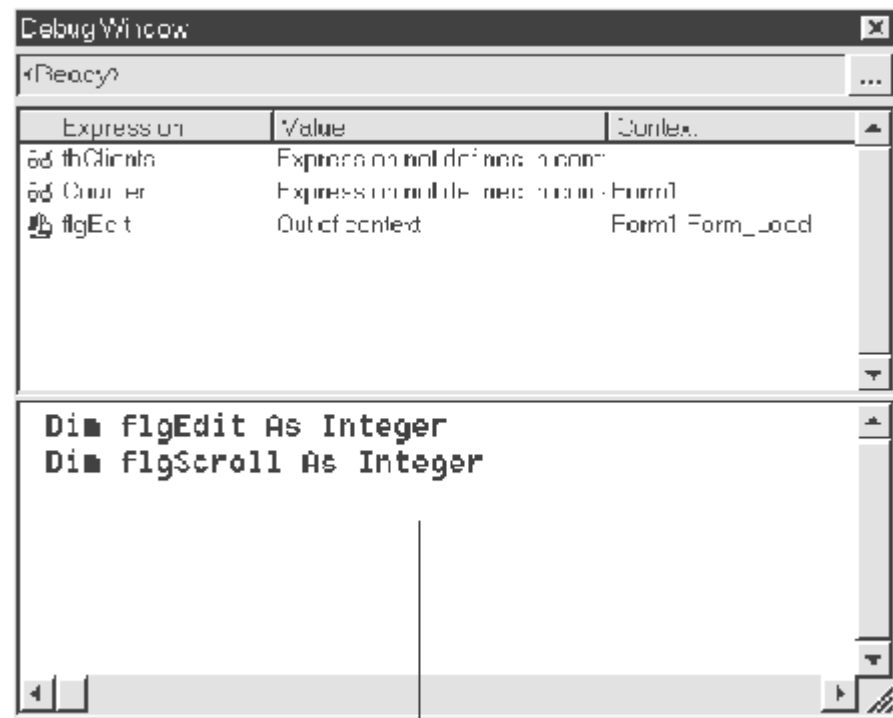
- **DATA_ERRDISPLAY** - это значение по умолчанию. Будет выведено стандартное сообщение об ошибке Microsoft Access.

Отладка программы в Visual Basic

Основным визуальным средством отладки в Visual Basic является окно **Debug**. Это окно позволяет выполнять достаточно много функций, и его основные элементы представлены на рис. 12.7.

Панель наблюдения за значениями

Кнопка программных вызовов



Панель отображения кода

Рис. 12.7. Окно отладки Debug в Visual Basic

Это окно имеет две панели. В верхней части расположена панель **Watch**, которая позволяет выводить значения интересующих нас переменных, значений свойств или выражений во время выполнения программы. Переменная или выражение могут быть занесены на эту панель с помощью команды **Add Watch** меню **Tools**. После выполнения этой команды на экране появляется одноименное диалоговое окно, представленное на рис. 12.8.



Заранее выделенное выражение автоматически помещается в поле **Expression** этого окна. В блоке **Context** определяется диапазон, в котором будет отслеживаться изменение наблюдаемой величины. Не увлекайтесь, попытка отследить значения во всем диапазоне работы программы (пункты **All Procedures** и **All Modules**) может резко замедлить выполнение программы. В блоке **Watch Type** вы можете задать способ реакции **Visual Basic** на изменение значения наблюдаемого выражения.

В нижней части окна отладки расположена панель **Immediate**, которая позволяет отображать информацию, связанную с отлаживаемыми операторами, а также непосредственно вводить необходимые команды, как это можно делать в окне **Command Visual FoxPro**.

Сама процедура выполнения отладки программы **Visual Basic** аналогична описанной ранее для программы **Access**.

12.3. Подготовка приложения для распространения

Если вы считаете, что разработанное вами приложение вполне подходит для работы пользователя, пора подумать о том, как наиболее удобно переместить его на другой компьютер, который к тому же вряд ли имеет соответствующую среду разработки.

В этом параграфе мы рассмотрим средства подготовки пользовательского приложения для распространения.

Современное приложение для обработки данных может представлять собой достаточно большое число различных модулей, включающих данные и программы для их обработки. В этом плане чрезвычайно большое количество файлов имеет приложение, разработанное на **Visual FoxPro**. С него и начнем.

В процессе создания пользовательского приложения по мере насыщения **Project Manager** различными модулями, **Visual FoxPro** автоматически создавал таблицу с расширением **PJX**, куда помещал необходимую информацию о составных элементах будущей прикладной программы. Используя эту таблицу, **Project Manager** может создать пользовательскую программу **Visual FoxPro**, которая будет включать в себя все элементы вашего проекта двух типов: в виде файла с расширением **APP** и в виде файла с расширением **EXE** (рис. 12.9). Для этого в **Project Manager** вам

следует просто нажать кнопку **Build**. Но чуть-чуть терпения, перед тем как вы нажмете эту кнопку, нам следует рассказать еще о некоторых моментах, на которые следует обратить внимание перед построением пользовательской программы.

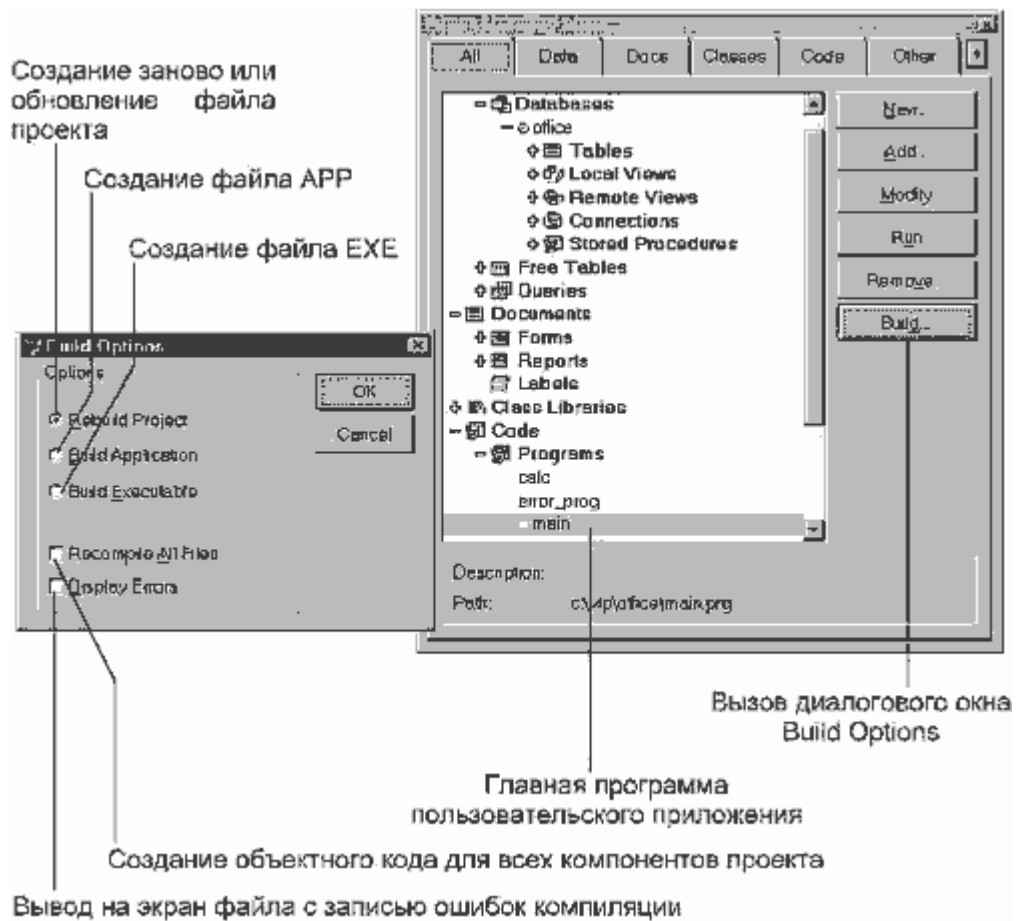


Рис. 12.9.

В Project Manager все внесенные в него файлы делятся на две группы: включенные в пользовательскую программу или не включаемые в нее. Например, по умолчанию не включаются в программу файлы БД и все связанные с ней файлы. Посмотрите на список файлов в Project Manager. Перед именем таких файлов вы обнаружите перечеркнутый кружок. Включаемый в пользовательскую программу файл компилируется в объектный код и недоступен для изменения. Файлы, не включаемые в пользовательскую программу, должны распространяться отдельно. Для того чтобы включить или исключить какой-то файл из пользовательской программы, в меню **Project** выберите команду **Project Info**. Появляющееся после этого диалоговое окно **Project Information** приведено на рис. 12.10. Для изменения статуса файла откройте вкладку **Files**. Таким образом, перед построением пользовательской программы вы должны проверить наличие всех файлов, которые должны войти в файл приложения. Например, файлы изображений автоматически не включаются в проект, и вы должны вручную добавить их, так как распространение их в виде отдельных файлов запрещено лицензионным соглашением.

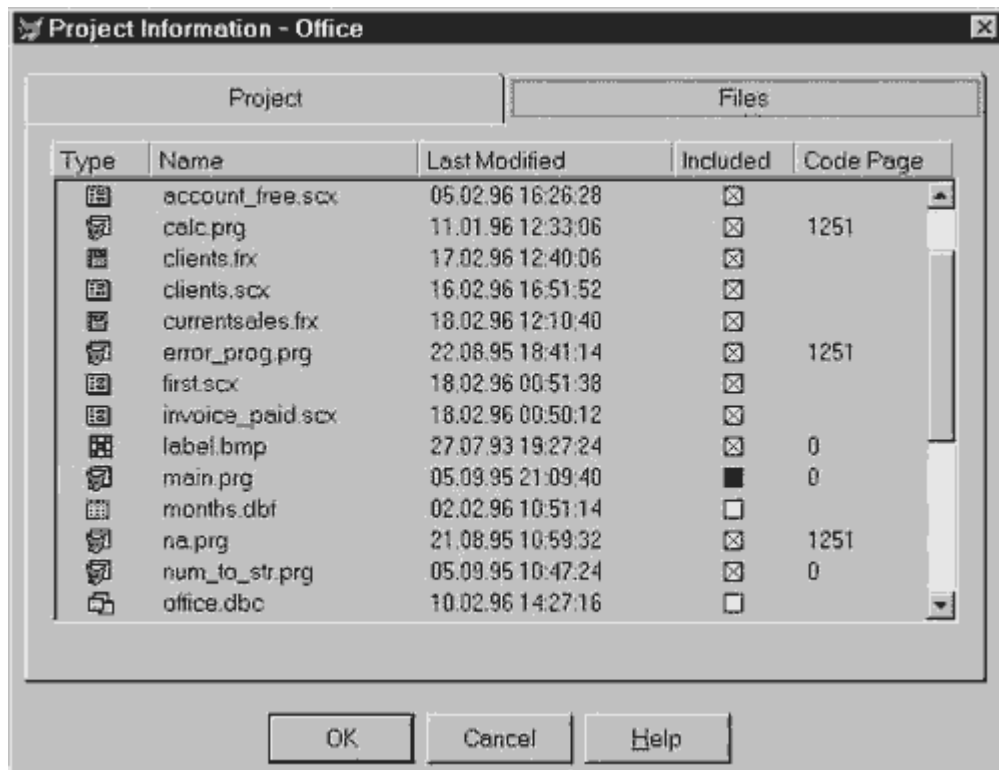


Рис. 12.10.

На вкладке **Project** того же окна, которая показана на рис. 12.11, мы можем указать авторские реквизиты, задать параметры компиляции и выбрать значок, который будет включен в EXE-файл и может использоваться для создания ссылки при его запуске. Для подготовки пользовательской программы в параметрах компиляции следует отменить опцию **Debug Info**, так как при включенной опции в объектный код помещается специальный отладочный модуль, позволяющий при ошибке с помощью окна *Trace* указать место ее возникновения при наличии исходного файла и самой СУБД. Так как в пользовательской программе исходных текстов программ нет, то этот модуль бесполезен и только занимает лишнее место.

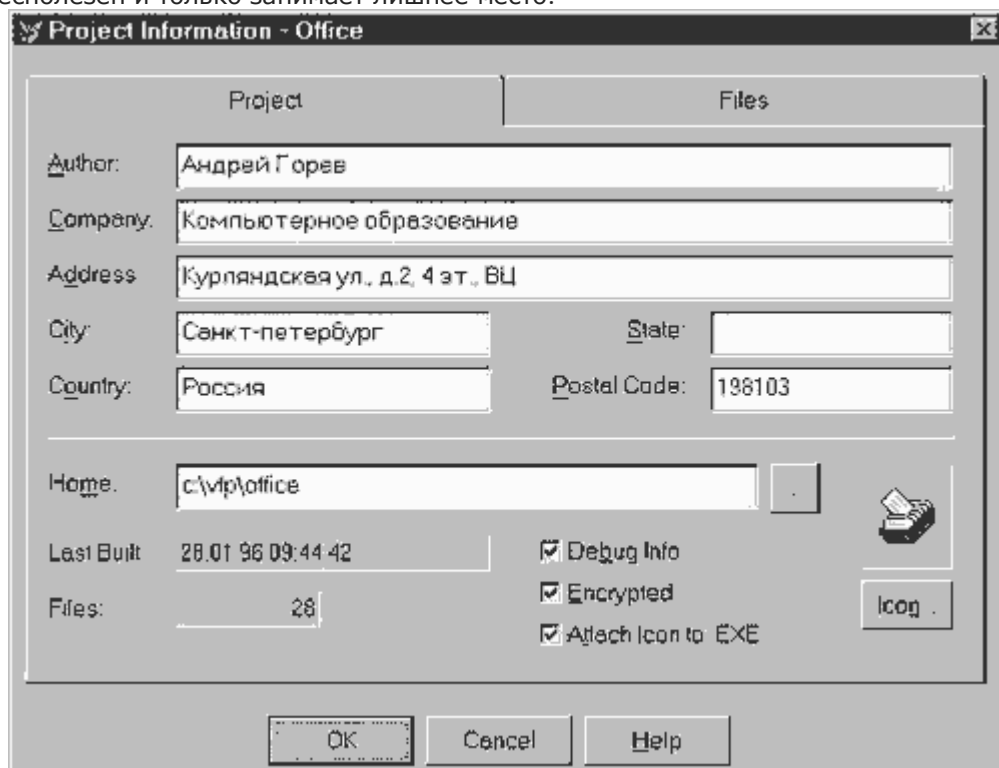


Рис. 12.11.

Вы можете также включить опцию **Encrypted**, которая исключает возможность просмотра символьных фрагментов в объектном коде. Правда, практика использования предыдущих версий **FoxPro** показывает, что это не слишком надежная защита от умелых хакеров. С другой стороны, программы преобразования файлов пользовательской программы в исходный текст несколько раз помогли авторам спасти собственные разработки, для которых непостижимым образом исчезали исходные файлы, а в пользовательской программе вдруг обнаруживалась ошибка.

Для того чтобы при компиляции установить точку запуска приложения, в **Project Manager** необходимо указать главную программу, как это видно на рис. 12.9. Для этого достаточно поставить курсор на этот файл и вызвать команду **Set Main** в меню **Project**. Как правило, в качестве главной программы используется или специальная программа, или файл главного меню приложения.

Для построения приложения в **Visual FoxPro** можно использовать и специальную программу **BUILDAPP.PRG**, которая при установке профессиональной версии **Visual FoxPro** записывается в папку **TOOLS\BUILDAPP** папки **Visual FoxPro**. Эта программа создает файл пользовательского приложения и удаляет из него исходный код событий и методов, который хранится в файлах форм **SCX** и визуальных библиотек **VCX**. Удаление исходного кода позволяет получить файл пользовательского приложения меньших размеров и более защищенный от вскрытия.

Для запуска программы используйте следующий синтаксис:

DO BUILDAPP [WITH *ProjectName* [, *AppFileName* [, *DebugMode* [, *BuildEXE*]]]]

Здесь параметр ***ProjectName*** задает имя проекта. Параметр ***AppFileName*** - имя файла пользовательского приложения, причем указываемое расширение (**APP** или **EXE**) будет устанавливать тип создаваемого файла. Параметр ***DebugMode*** имеет логический тип и позволяет включить или отключить отладочный режим (для **SET DEBUG ON** параметр ***DebugMode*** должен быть равен **.T.**, а для **SET DEBUG OFF** - **.F.**). Если параметр ***BuildEXE*** равен **.T.**, то будет создаваться выполняемый **EXE**-файл.

Программа **BUILDAPP** выполняет следующие действия:

1. Определяет местоположение файлов.
2. Открывает проект.
3. Создает файл пользовательского приложения на основе первоначального файла проекта для проверки того, что весь исходный код откомпилирован и объектный код записан в соответствующие поля файлов **SCX** и **VCX**. Для завершения работы программы проект не должен генерировать ошибок во время построения пользовательского приложения.
4. Физически удаляет помеченные для удаления записи в файлах **VCX**.
5. Переносит исходный код из файлов **SCX** и **VCX** в массив.
6. Перестраивает файл пользовательского приложения.
7. Восстанавливает исходный код в файлах **SCX** и **VCX**, возвращая тем самым проект в первоначальное состояние.

Теперь скажем несколько слов о двух типах пользовательской программы, которые можно создать в **Visual FoxPro**.

APP-файл предназначен для работы в среде СУБД. Таким образом, компьютер, на котором будет работать ваша программа, должен быть оснащен копией **Visual FoxPro**.

EXE-файл на несколько десятков килобайт больше **APP**-файла, но для своей работы не требует наличия на компьютере **Visual FoxPro**. Этот файл работает совместно с библиотекой поддержки **VFP300.ESL**. **EXE**-файл работает несколько быстрее, чем **APP**, вероятно, за счет того, что требует меньшего количества ресурсов.

При выборе типа файла для пользовательского приложения необходимо иметь в виду, что библиотека поддержки не включает некоторые функции СУБД. Следовательно, в приложении, распространяемом в виде **EXE**-файла, не должны присутствовать эти исключенные функции.

Из меню **Visual FoxPro** не доступны следующие возможности:

- Database
- Form
- Menu
- Program
- Project
- Query
- Table

Ниже приведен список команд, при выполнении которых в пользовательском приложении произойдет ошибка "Feature not available" (функция не доступна).

Команды, не доступные при использовании библиотеки поддержки:

BUILD APP MODIFY FORM
 BUILD EXE MOFIFY MENU
 BUILD PROJECT MODIFY PROJECT
 COMPILE MODIFY QUERY
 CREATE FORM MODIFY STORED PROCEDURE
 CREATE MENU MODIFY STRUCTURE
 CREATE QUERY MODIFY VIEW
 CREATE VIEW SUSPEND
 FILER SET
 MODIFY CONNECTION SET STEP
 MODIFY DATABASE

В следующем списке перечислены файлы, которые не могут распространяться путем включения в пользовательское приложение или вместе с ним.

ADDLABEL.APP
 AUTONAME.PRG
 BROWSER.APP
 BUILDAPP.PRG
 BUILDAPP.SCT
 BUILDAPP.SCX
 BUILDER.APP
 CONPROCS.PRG
 CONVERT.APP
 CONVERT.H
 CONVERT.PJT
 CONVERT.PJX
 CONVERT.PRG
 CVTALERT.H
 CVTSCX.H
 FD3.FLL
 FDKEYWRD.CDX
 FOREIGN.H
 FOREIGN.PRG
 FORMPARM.PRG
 FOXHELP.DBF
 FOXHELP.FPT
 FOXHELP.HLP
 FPCNEW.PRG
 GENDBC.PRG
 GENERIC.PRG
 GENMENU.PRG
 HC35.ERR
 HC35.EXE
 IMAGEDIT.EXE
 IMAGEDIT.HLP
 JD.FKY
 JD.PRG.
 LOCWORD.H
 LOCWORD.PRG
 MIGDB4.H
 MIGDB4.PRG
 MIGNAVPR.TXT
 MMSETUP.PRG
 MRBC.EXE
 MSGRAPH.HLP
 MSINFO.EXE
 PRO_EXT.H
 PUTNAME.PRG
 RESERVED.FLL
 SHED.EXE

SHED.HLP
 SPELLCHK.APP
 TRANSPRT.PRG
 VFP.EXE
 WINAPIMS.LIB
 WIZARD.APP
 WIZARD.FLL
 WZFORM.APP
 WZFOXDOC.APP
 WZGRAPH.APP
 WZIMPORT.APP
 WZMAIL.APP
 WZPIVOT.APP
 WZQUERY.APP
 WZREPORT.APP
 WZSETUP.APP
 WZTABLE.APP
 WZUPSIZE.APP

В Visual Basic процедура подготовки пользовательского приложения очень похожа на только что описанную для Visual FoxPro. В Visual Basic мы можем получить только EXE-файл. Для этого достаточно выбрать команду *Make EXE File* в меню *File*. Подготовка дистрибутивного комплекта дискет в Visual Basic выполняется с помощью отдельного приложения - **Application Setup Wizard**, представленного на рис. 12.12. Это приложение работает аналогично Setup Wizard в Visual FoxPro и путем выполнения семи шагов позволяет программисту быстро определить набор файлов, необходимый для работы приложения на компьютере пользователя.

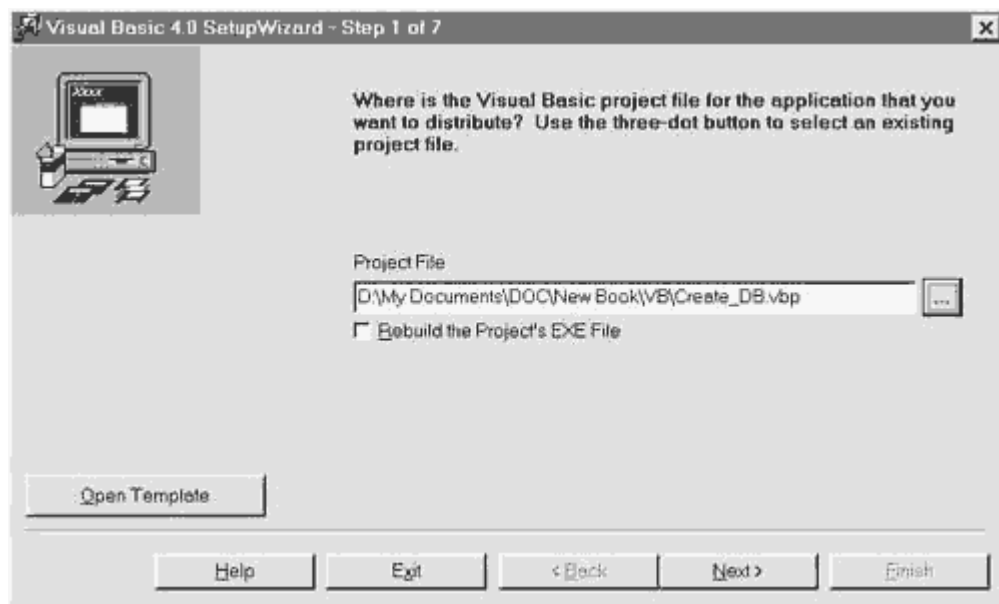


Рис. 12.12.

Приложение 1

Дополнительные возможности новой версии Visual FoxPro 5.0

Визуальные средства проектирования
 Поставка программного пакета
 Требования к установке
 Project Manager

Работа с кодом программы
 Создание базы данных
 Работа с данными
 Расширение возможностей технологии клиент-сервер
 Построение пользовательского интерфейса
 Расширение функций OLE
 Отладка приложения

К тому времени, когда рукопись данной книги уже была сдана в издательство, вышла новая версия СУБД Visual FoxPro, которая получила номер версии 5.0. Где же версия 4? Ведь в книге все время говорилось о Visual FoxPro версии 3.0. Ничего страшного не произошло? и разработчики Microsoft еще не разучились считать. Visual FoxPro получил номер версии 5 для унификации с новыми версиями других средств разработки Microsoft: Visual Basic 5.0 и Visual C++ 5.0. Ну что ж, это является лишним свидетельством постоянной тенденции к унификации средств разработки для совершенствования процесса создания прикладного программного обеспечения.

В связи с появлением на полках магазинов новой версии Visual FoxPro мы решили кратко рассказать о тех новых возможностях, которые появились у программистов по сравнению с изложенными в книге.

Визуальные средства проектирования

В новой версии Visual FoxPro расширен набор визуальных средств разработки приложений. Появились два новых Мастера.

Мастер создания приложения (Application Wizard) позволяет объединить возможности других Мастеров Visual FoxPro и в то же время обладает новыми уникальными функциями. На рис. П.1.1 приведен вид этого Мастера на первом шаге. Вы можете выбрать один из вариантов дальнейших действий:

- Создание проекта, структуры папок для размещения входящих в проект файлов и нескольких исходных модулей на основе базовых классов Visual FoxPro. Для этого необходимо, как это видно на рис. П.1, выбрать опцию **Framework only**. При выборе этого варианта работа Мастера заканчивается на втором шаге. Вы получаете файл проекта с двумя заготовками форм, две визуальные библиотеки классов, меню и главную программу для запуска приложения.
- Дополнительно к первому варианту создание полнофункционального приложения с набором, по вашему выбору, создаваемых форм и отчетов. При этом вы должны указать базу данных, для которой будет генерироваться приложение.

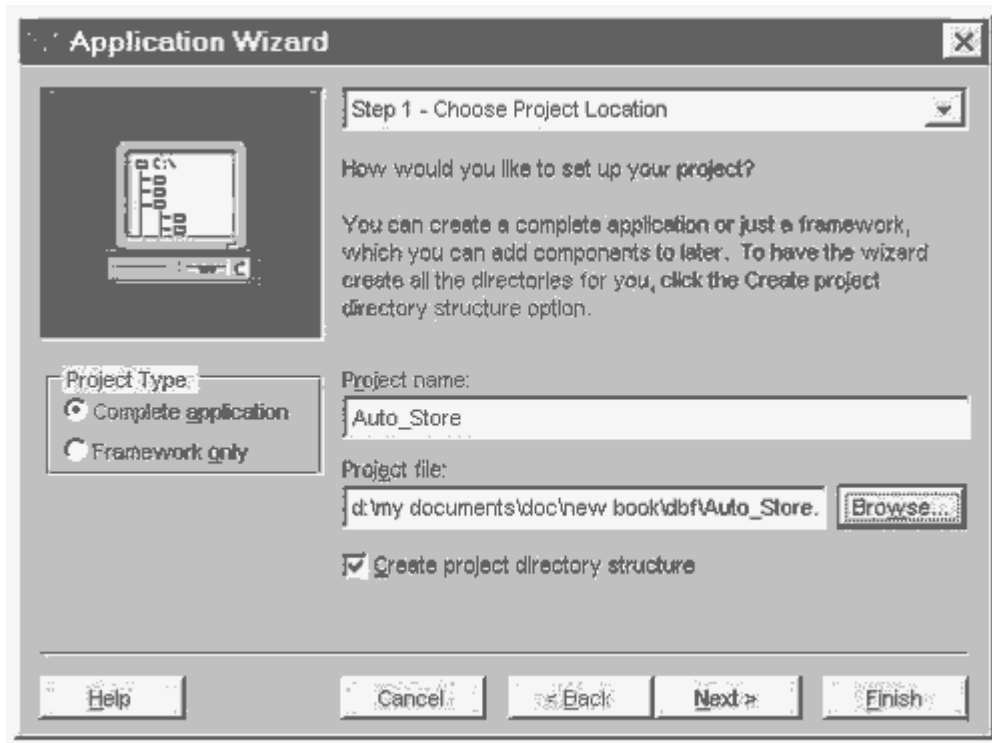


Рис. П.1.1. Первый шаг работы с Мастером создания приложения

Мастер наращивания для СУБД Oracle Server 7.0 (Oracle Upsizing Wizard) позволяет перенести данные из локальной БД на сервер Oracle, преобразовать локальные таблицы и представления в таблицы и представления внешней БД и, насколько это возможно, продублировать функциональность приложения Visual FoxPro для БД Oracle. Обратите внимание, что поставляемый драйвер ODBC для Oracle работает только в ОС Windows NT. На компьютере должна быть установлена копия программы SQL Net - составной части клиентского программного обеспечения Oracle.

Расширена функциональность и удобство использования существующих Мастеров. Основные нововведения:

- Form Wizard позволяет теперь создавать страничные блоки для увеличения числа размещаемых полей.
- Pivot Table Wizard и Mail Merge Wizard позволяют теперь использовать 32-разрядный драйвер ODBC Visual FoxPro.
- Upsizing Wizard поддерживает установленные правила ссылочной целостности.

Свободно распространяемый ранее **Мастер для создания Web-страниц (World Wide Web Search Page Wizard)** теперь включен в состав пакета как утилита, которая позволяет на основе записей, хранящихся в БД, создать Web-страницы для поиска и отображения данных.

Поставка программного пакета

Visual FoxPro 5.0 поставляется только в виде профессиональной версии на CD-ROM, которая содержит все необходимые элементы для создания пользовательского приложения. Внешне коробка с пакетом программ теперь существенно "похудела" и напоминает коробку с Visual C++. Это произошло за счет того, что объемная печатная документация заменена на мощную интерактивную среду Online Documentation. С ее помощью вы можете искать необходимый материал, пользоваться перекрестными ссылками и даже смотреть мультимедиа клипы о способах наиболее эффективной работы с Visual FoxPro. Интерфейс электронной документации очень удобен, но... А "но" заключается в том, что физически электронная документация представляет собой файл БД Access размером около 70 Мбайт плюс примерно такой же объем файлов с видеоклипами. А это значит, что для работы с ней вам потребуется ну очень мощный компьютер. По крайней мере, 486DX2-66 с 20 Мбайт ОЗУ и очень быстрым жестким диском с этой задачей не справился.

Требования к установке

Новая версия Visual FoxPro не будет работать на 16-разрядной платформе Windows 3.x. Для работы самой СУБД и разработанного на ее основе приложения требуется Windows 95, Windows NT 3.51 или Windows NT 4.0.

Корпорация Microsoft так определяет минимальные требования к компьютеру для работы Visual FoxPro 5.0 под управлением ОС Windows 95:

- Процессор 486 с частотой 50 MHz.
- Мышь.
- Объем ОЗУ 10 Мбайт.
- Свободное пространство на жестком диске: 15 Мбайт для установки варианта для переносного компьютера, 100 Мбайт - для типичной установки и 240 Мбайт - для полной.

Для работы Setup Wizard и создания OLE Automation сервера на компьютере должна быть установлена Runtime версия Visual FoxPro. Проверьте, не исключили ли вы этот пункт при установке Visual FoxPro.

Project Manager

Теперь в Project Manager ориентироваться стало значительно легче за счет того, что каждый объект сопровождается значком, идентифицирующим его тип (рис. П.1.2). Более тесной стала интеграция с Microsoft Visual SourceSafe - пакетом программ для организации коллективной работы над пользовательским приложением. Project Manager будет визуально отображать статус объекта.

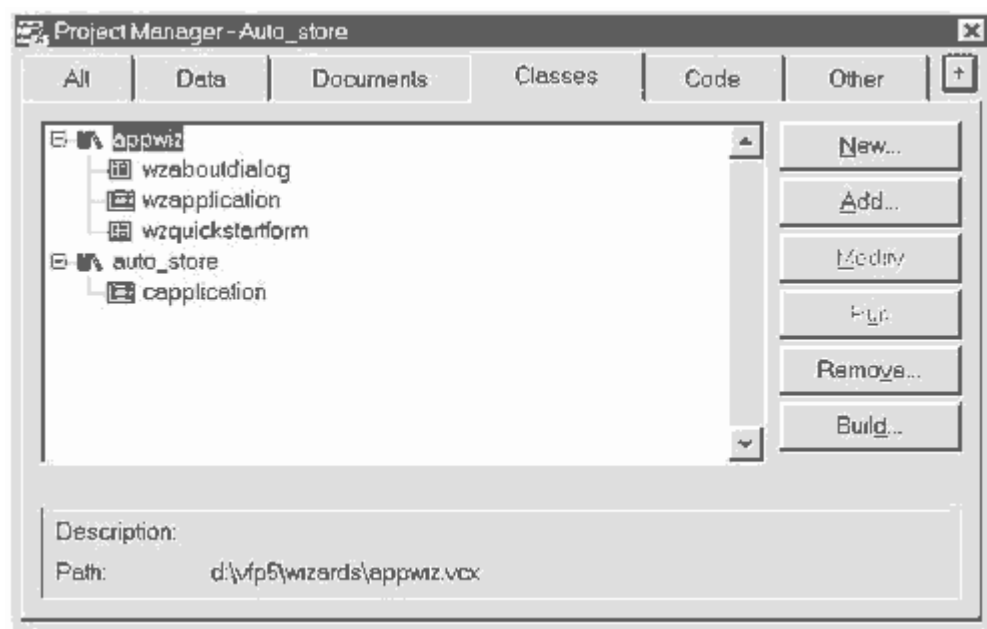


Рис. П. 1.2. Отображение типа объекта в Project Manager

Работа с кодом программы

Существенные усовершенствования внесены в процесс разработки программ. Наконец-то появился цвет в Редакторе и, соответственно, в окне *Command*, так, как это реализовано в Visual Basic. Работая над кодом программы, нажмите правую кнопку мыши, и вы увидите длинное контекстное меню, изображенное на рис. П.1.3.

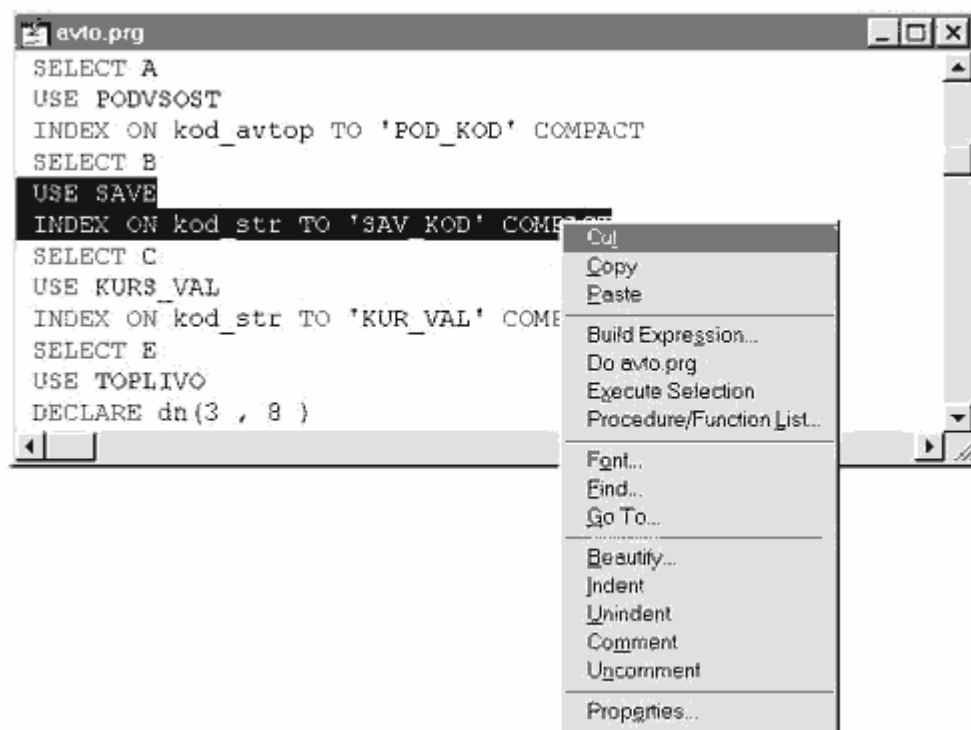


Рис. П.1.3.

При выборе команды *Build Expression* можно воспользоваться Построителем выражений. Очень удобно, особенно если лень разбираться со сложным синтаксисом. Теперь мы можем выполнять не только программу, но и ее произвольный фрагмент, выбрав команду *Execute Selection*. Следующая команда *Procedure/Function List* выводит диалоговое окно (рис. П.1.4), в котором, используя список процедур и функций вашей программы, можно быстро перейти к требуемому фрагменту.

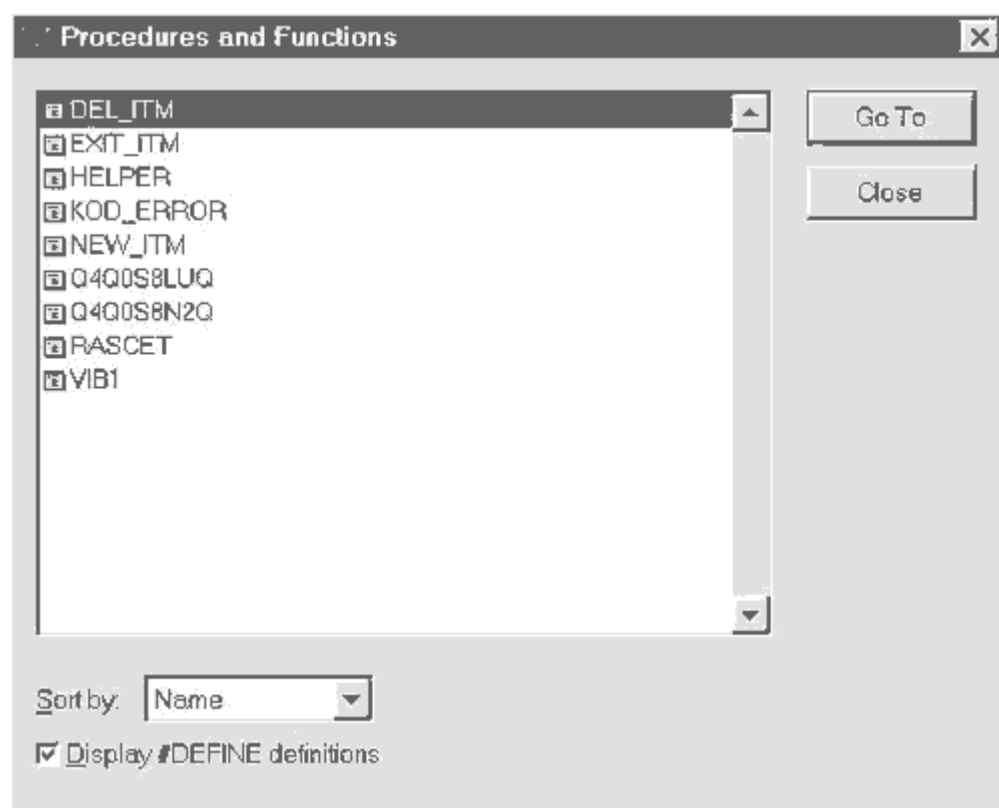


Рис. П.1.4.

В контекстном меню, приведенном на рис. П.1.3, вы найдете еще две новые команды: *Comment* и *Uncomment*. Эти команды позволяют мгновенно пометить как комментарий выделенный фрагмент кода, который не нужно в данный момент выполнять, и, наоборот, убрать установленные ранее символы комментария. Чрезвычайно полезная возможность в период отладки программы.

Создание базы данных

В новой версии контейнер БД существенно расширен. Особенно ярко это проявляется при проектировании таблиц. На рис. П.1.5 представлен новый Конструктор таблицы и отмечены вновь появившиеся элементы. Теперь для того, чтобы задать обычный индекс, совсем не обязательно переходить на другую вкладку - это можно сделать сразу при определении соответствующего поля.

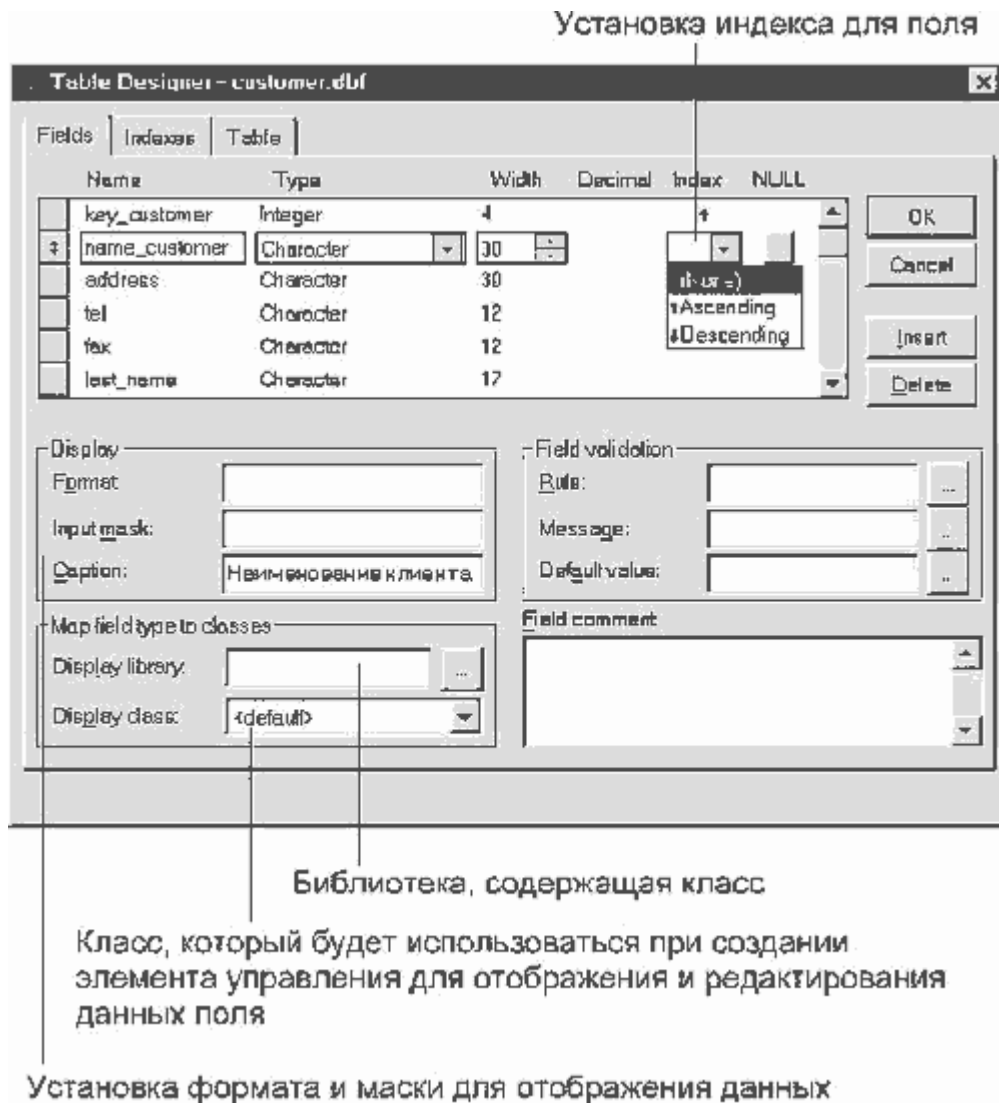


Рис. П.1.5.

Очень важным нововведением является возможность задать для поля класс элемента управления, с помощью которого будут отображаться данные при работе с формой. Еще более важным обстоятельством является возможность выбора этого класса из библиотеки разработчика. Такая возможность существенно облегчает процесс разработки пользовательского интерфейса. Новая вкладка *Table* позволяет быстро перейти к установке правил проверки и триггеров на уровне таблицы и записи.

Работа с данными

Развитие визуальных средств разработки приложений коснулось расширения возможностей программиста при создании представлений и запросов. Вы можете создавать внешние объединения, указывать псевдонимы для колонок, определять явно или в процентах число строк, помещаемых в результат.

Эти возможности основываются на расширенном синтаксисе команды **SELECT-SQL**:

```
SELECT [ALL | DISTINCT] [TOP nExpression [PERCENT]]
      [Alias.] Select_Item [AS Column_Name]
      [, [Alias.] Select_Item [AS Column_Name] ...]
FROM [FORCE]
     [DatabaseName!] Table [Local_Alias]
     [[INNER | LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER] JOIN
     DatabaseName!] Table [Local_Alias]
     [ON JoinCondition]]
[[INTO Destination] | [TO FILE FileName [ADDITIVE] | TO PRINTER [PROMPT] | TO SCREEN]]
[PREFERENCE PreferenceName]
[NOCONSOLE]
[PLAIN]
[NOWAIT]
[WHERE JoinCondition [AND JoinCondition ...] [AND | OR FilterCondition [AND | OR FilterCondition
...]]]
[GROUP BY GroupColumn [, GroupColumn ...]]
[HAVING FilterCondition]
[UNION [ALL] SELECT Command]
[ORDER BY Order_Item [ASC | DESC] [, Order_Item [ASC | DESC] ...]]
```

Опция INTO *Destination* при направлении результата запроса в курсор получила новое предложение

CURSOR *CursorName* [NOFILTER]

Использование опции **NOFILTER** позволяет облегчить создание последовательных запросов, так как в этом случае не надо указывать константы в результате первого запроса для резервирования колонок. Курсор, созданный с этой опцией, записывается в виде временной таблицы на диск, которая уничтожается при закрытии курсора.

Используя Конструктор представления можно указать для представления такие же расширенные свойства, как ранее были возможны только для таблицы (формат, маска ввода и т. д.).

Расширение возможностей технологии клиент-сервер

В связи с тем, что в настоящий момент Visual FoxPro является одним из наиболее эффективных средств разработки клиентских программ для приложений, работающих по технологии клиент-сервер, развитию этих возможностей в новой версии было уделено большое внимание.

Первое, что бросается в глаза - это более глубоко интегрированный в Администратор ODBC Конструктор соединения (Connection Designer). Он также обеспечивает возможность установки дополнительных свойств для оптимизации соединений.

Для повышения гибкости работы с внешним сервером Visual FoxPro 5.0 приобрел новый элемент работы с данными - **Offline Views** - независимые представления. Что это такое? Это представления, которые после получения данных могут использоваться самостоятельно, в отрыве от источника данных, и при необходимости обновлять данные в источнике на основе выполненных пользователями изменений.

Независимые представления могут использоваться, например, в ситуации, когда с приложением работает пользователь, который в силу специфики своих обязанностей использует переносной компьютер. В этом случае он может в начале рабочего дня с помощью независимых представлений создать подмножество БД, хранящейся на сервере. В конце рабочего дня, заехав в офис, он может обновить БД на сервере данными, которые были изменены в независимых представлениях в течение рабочего дня. При обновлении данных Visual FoxPro автоматически управляет задачей координации изменений между независимым представлением и данными в БД.

Для создания и использования независимых представлений вы можете применять следующие команды и функции.

Для получения независимого представления на основе существующего откройте соответствующую БД и используйте функцию

CREATEOFFLINE(*ViewName* [, *cPath*])

которая возвращает .Т., если независимое представление успешно создано. Параметр *ViewName* указывает имя существующего представления. Параметр *cPath* позволяет указать папку для расположения временных таблиц для независимого представления.

Открыть и работать с данными в независимом представлении можно, используя команду

```
USE [[DatabaseName.]Table | SQL ViewName | ?]
[IN nWorkArea | cTableAlias]
[ONLINE]
[ADMIN]
[AGAIN]
[NOREQUERY [nDataSessionNumber]]
[NODATA]
[INDEX IndexFileList | ?]
[ORDER [nIndexNumber | IDXFileName | [TAG] TagName [OF CDXFileName] [ASCENDING |
DESCENDING]]]
[ALIAS cTableAlias]
[EXCLUSIVE]
[SHARED]
[NOUPDATE]
```

Новая опция **ADMIN** позволяет просмотреть все выполненные изменения и при необходимости отменить какие-либо изменения.

Опция **ONLINE** используется для переноса изменений, выполненных в независимом представлении на БД сервера.

Для соединения независимого представления с БД сервера и перехода в режим работы обычного представления выполните функцию

DROPOFFLINE(*cViewName*)

Построение пользовательского интерфейса

Несмотря на то, что внешне в новой версии вы найдете мало различий в визуальных средствах разработки пользовательского интерфейса, как только вы начнете работать, сразу столкнетесь с порой пусть небольшими, но приятными нововведениями.

Конструктора формы коснулись следующие улучшения:

- Поддерживается как однодокументный (SDI), так и многодокументный (MDI) интерфейс.
- Расширена возможность использования технологии перетаскивания, теперь, перетаскивая с помощью мыши поля таблиц на проектируемую форму, для создания объекта используется класс, указанный в контейнере БД.
- Появилась возможность просматривать и устанавливать требуемое значение свойства сразу для нескольких элементов управления, имеющих одинаковые свойства.
- Работая в окне **Properties**, вы можете переключать активные элементы управления нажатием клавиш **Tab**, **PgDn**, **PgUp**, **Home** и **End** совместно с клавишей **Ctrl**. Без клавиши **Ctrl** эти клавиши помогут вам быстро находить требуемые свойства в списке.
- Многие диалоговые окна остаются на экране и облегчают тем самым множественный выбор, не требуя многократного вызова.
- На стандартной панели инструментов новая кнопка **Design**, наряду с уже существовавшей в третьей версии кнопкой **Run**, позволяет быстро переключаться между режимом работы с формой в режим проектирования и обратно.

В новой версии **Visual FoxPro** при создании нового файла меню вам будет предложен выбор между созданием привычного меню и контекстного меню (**shortcut menu**), вызываемого при нажатии правой кнопки мыши на определенном элементе управления. Для этого достаточно для события **PightClick** соответствующего элемента управления записать код запуска файла требуемого меню:

DO edtshort.mpr

В Конструкторе меню появилась новая кнопка **Insert Bar**, которая позволяет вывести диалоговое окно со списком стандартных команд главного меню **Visual FoxPro** и разместить нужную команду в вашем меню. Теперь не придется искать в документации мудреное название команды *Copy* из меню *Edit*, если вы решили предоставить пользователю возможность копировать данные.

Значительно расширен набор примеров (**Solutions**), в которых показано, как можно наиболее эффективно использовать преимущества **Visual FoxPro** при решении конкретных задач, с которыми программист сталкивается при работе над пользовательским приложением. Эти примеры можно использовать как отдельные компоненты вашего будущего приложения.

Расширение функций OLE

Основная новость здесь - **Visual FoxPro** стал OLE-сервером. Таким образом, другие приложения, поддерживающие стандарт OLE 2.0, могут использовать объекты **Visual FoxPro** для расширения своей функциональности. Для этих целей вы можете создать как out-of-process (EXE), так и in-process (DLL) сервер.

В **Visual FoxPro 5.0** доступ к объектам выполняется, как и в подавляющем большинстве других OLE-серверов, с помощью объекта верхнего уровня **Application**. В табл. П.1.1 перечислены его свойства, а в табл. П.1.2 - методы.

Таблица П.1.1. Свойства объекта **Application**

Свойство	Параметры	Описание
ActiveForm.Property [= <i>Setting</i>]	<i>Property</i> - свойство формы	Обеспечивает ссылку на активную форму или объект _SCREEN
ActiveForm.Method	Setting - значение свойства <i>Method</i> - метод формы	
AutoYield [= <i>IExpr</i>]	<i>IExpr</i> - по умолчанию равен .T., что предусматривает приоритет событий Windows . Значение .F. предотвращает прерывание выполнения кода Visual FoxPro . При этом события Windows ставятся в очередь	Определяет способ обработки событий Windows
Caption [= <i>cText</i>]	<i>cText</i> - текст заголовка	Определяет заголовок окна приложения
DefaultFilePath [= <i>cPath</i>]	<i>cPath</i> - обозначение устройства, папки или пути	Определяет папку по умолчанию для приложения
FullName		Определяет путь и имя файла для запуска копии Visual FoxPro
Height [= <i>nHeight</i>]	<i>nHeight</i> - высота окна приложения	Определяет высоту окна приложения
Left [= <i>nDist</i>]	<i>nDist</i> - расстояние от левого края	Определяет расположение окна приложения с левого

<i>Name</i> [= <i>cName</i>]	<i>cName</i> - имя объекта	края Задает имя объекта для ссылки в коде программы
<i>OLERequestPendingTimeout</i> [= <i>nMilliseconds</i>]	<i>nMilliseconds</i> - величина задержки в миллисекундах. По умолчанию равна 5000 мс. Если параметр равен 0, сообщение не появляется	Определяет задержку времени, которая происходит перед появлением сообщения о том, что система занята в процессе выполнения запроса OLE Automation, если пользователь использует клавиатуру или мышь
<i>OLEServerBusyRaiseError</i> [= <i>lExpression</i>]	<i>lExpression</i> - по умолчанию равен .F. и сообщение об ошибке будет появляться. Если параметр равен .T., сообщения не будет	Определяет появление сообщения об ошибке, когда истечет время, установленное в свойстве <i>OLEServerBusyTimeout</i>
<i>OLEServerBusyTimeout</i> [= <i>nMilliseconds</i>]	<i>nMilliseconds</i> - величина времени в миллисекундах до появления сообщения о том, что сервер занят	Определяет время, в течение которого происходит повторное выполнение запроса OLE Automation, если занят сервер
<i>StartMode</i>		Возвращает число, идентифицирующее тип запускаемого приложения
<i>StatusBar</i> [= <i>cMessageText</i>]	<i>cMessageText</i> - строка сообщения	Определяет текст в статус-строке приложения
<i>Top</i> [= <i>nDist</i>]	<i>nDist</i> - расстояние от верхнего края	Определяет расположение окна приложения от верхнего края
<i>Version</i>		Возвращает в виде строки символов номер версии запускаемого приложения
<i>Visible</i> [= <i>lExpr</i>]	<i>lExpr</i> - по умолчанию равен .F., то есть запускаемая копия приложения не видима. Если значение равно .T. - приложение становится видимым	Определяет, будет ли запускаемая копия приложения видима
<i>Width</i> [= <i>nWidth</i>]	<i>nWidth</i> - ширина	Определяет ширину

	окна приложения	окна приложения
Таблица П.1.2. Методы объекта Application		
Метод	Параметры	Описание
DataToClip (<i>nWorkArea</i> <i>cTableAlias</i>) [, <i>nRecords</i>] [, <i>nClipFormat</i>])	<i>nWorkArea</i> , <i>cTableAlias</i> - рабочая область или псевдоним источника данных. <i>nRecords</i> - число копируемых записей. <i>nClipFormat</i> - по умолчанию равен 1, при этом данные полей разделяются пробелами. Если параметр равен 3, данные разделяются знаком табуляции	Копирует записи в буфер обмена в виде текста, в котором каждая запись занимает отдельную строку
DoCmd (<i>cCommand</i>)	<i>cCommand</i> - выражение, представляющее команду Visual FoxPro	Позволяет выполнить команду Visual FoxPro из приложения, являющегося OLE-контроллером
Eval (<i>cExpression</i>)	<i>cExpression</i> - выражение, которое необходимо преобразовать	Преобразует выражение и возвращает его в Visual FoxPro
Help ([<i>cFileName</i>] [, <i>nContextID</i>] [, <i>cHelpTopic</i>])	<i>cFileName</i> - имя и путь к файлу справки. <i>nContextID</i> - идентификатор раздела. <i>cHelpTopic</i> - тема раздела	Открывает окно с контекстной справкой
Quit ()		Закрывает запущенную копию приложения Visual FoxPro
RequestData ([<i>nWorkArea</i> <i>cTableAlias</i>] [, <i>nRecords</i>])	<i>nWorkArea</i> , <i>cTableAlias</i> - рабочая область или псевдоним источника данных. <i>nRecords</i> - число копируемых записей	Создает массив с данными из источника данных Visual FoxPro

Для ссылки на объект Application можно использовать системную переменную `_VFP`.

Visual FoxPro 5.0 имеет следующие коллекции, которые ассоциируются с объектом Application. Каждая коллекция может иметь соответствующие объекты:

- **Forms** - формы;
- **Objects** - объекты;
- **Controls** - элементы управления;
- **Page** - страницы;
- **Buttons** - кнопки;
- **Columns** - колонки.

При этом обратите внимание, что эти коллекции являются коллекциями исключительно OLE-объектов и могут использоваться только с объектом **Application**. К этим коллекциям нельзя обращаться, используя ассоциированные с включенными в них объектами переменные. Вы должны использовать свойство **Application** как это показано ниже:

```
oFrm = CREATEOBJECT('Form')
? oFrm.Application.Forms[1].Controls.Count
```

Отладка приложения

Новая версия Visual FoxPro позволяет более легко отлаживать и наблюдать за работой приложения. Вместо специальных диалоговых окон **Trace** и **Debug** теперь используется интегрированный отладчик **Debugger**. Новый отладчик может загрузаться в отдельное окно, не привязанное к главному окну Visual FoxPro или пользовательского приложения, что облегчает его использование и не мешает обычной работе. Новый отладчик просто прекрасен, а его основные возможности продемонстрированы на рис. П.1.6.

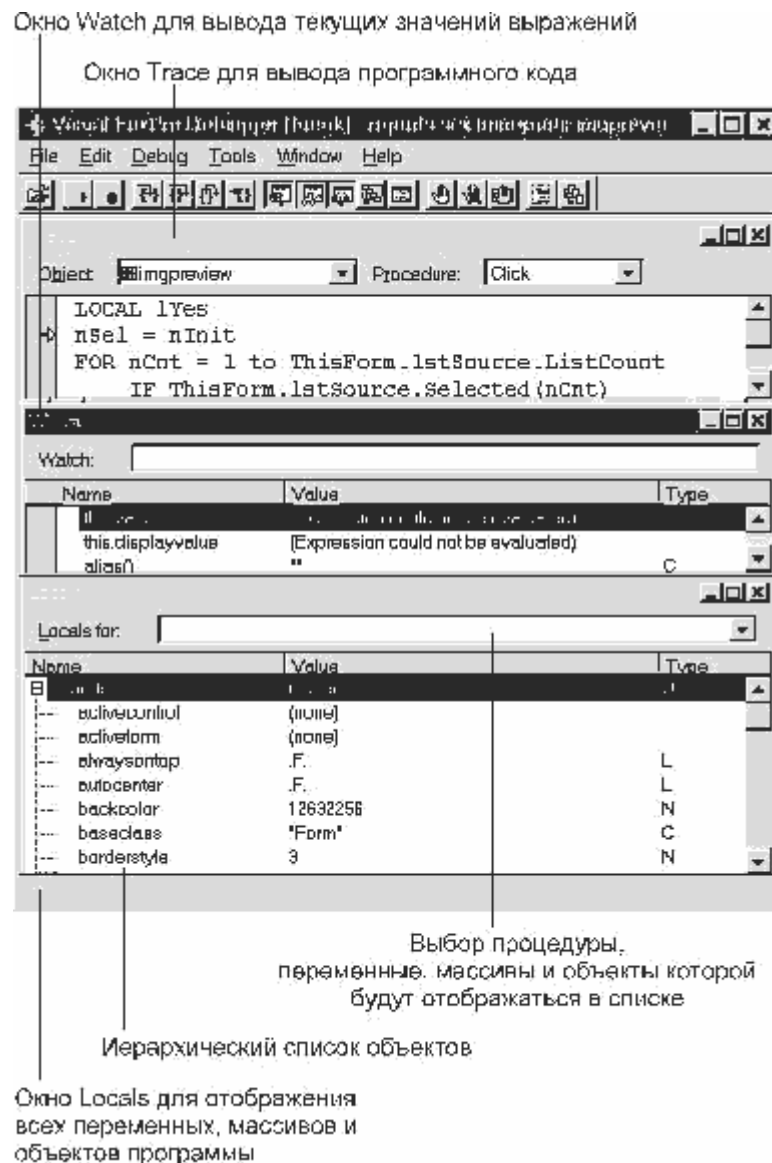


Рис. П.1.6.

Отладчик имеет несколько окон, обеспечивающих выполнение разнообразных функций.

В окне **Trace** вы можете просматривать программный код. Если отлаживается работа формы, то нужный фрагмент кода легко найти, выбирая из раскрывающихся списков в верхней части этого окна объект и событие или метод. Зона слева от программного кода зарезервирована для следующих символов:

- текущая выполняемая линия кода;
- **I** активная точка останова;
- **m** неактивная точка останова;
- позиция вызова в стеке, если вы проверяете выполнение кода на уровне, отличном от текущего выполняемого кода.

Двойной щелчок мыши в этой зоне позволяет установить или убрать точку останова выполнения программы на данной строке кода.

Окно **Watch** позволяет просматривать значения выражений. Набирая в текстовом поле соответствующие выражения и нажимая клавишу **Enter**, вы помещаете их в список, в котором отображается текущее значение и его тип. В зоне слева вы можете установить или снять точку останова, которая будет прерывать выполнение программы каждый раз при изменении значения выражения. Для включения в список сложных выражений нет необходимости набирать их заново. Выделите нужный текст в любом окне **Visual FoxPro** и перетащите его в окно **Watch**. Для редактирования выражения дважды щелкните на нем мышкой.

Помимо трех видимых окон, отладчик имеет еще два окна, которые могут вызываться из его меню **Window**:

- **Call Stack** - позволяет просматривать список выполняемых процедур, программ и методов;
- **Output** - позволяет просматривать выводимые данные активной программы, процедуры или метода.

Приложение 2

Взаимозаменяемость команд и функций Visual FoxPro и Visual Basic

В нашей книге мы старались показать возможности разработки приложений для обработки данных с использованием современных средств разработки фирмы **Microsoft**. С точки зрения программиста речь идет об использовании одной из двух платформ: **Visual FoxPro** или **Visual Basic**. На последней строится работа с **MS Access**, и обе платформы позволяют вполне успешно использовать архитектуру клиент-сервер на базе **MS SQL Server**, а также возможности **Windows API**. В книге речь шла и о совместном использовании различных платформ для решения тех или иных задач.

Для того чтобы облегчить читателям перенос своих приложений на другую платформу, мы включили в книгу данное приложение. Как уже неоднократно подчеркивалось, **Visual FoxPro** предоставляет инструментарий, узконаправленный для разработки приложений по обработке данных, в то время как **Visual Basic** является универсальным пакетом программирования. В связи с этим, естественно, что **Visual FoxPro** имеет более богатые средства работы с данными. При переносе приложения с **Visual FoxPro** на **Visual Basic** программист может столкнуться с дефицитом языковых средств, поэтому за сравнимую базу взят именно набор команд и функций **Visual FoxPro**.

В следующем списке приведено соответствие команд и функций **Visual Fox-Pro 5.0** с **Visual Basic**:

Команды и функции Visual FoxPro

Сравнение с Visual Basic

```
#DEFINE ... #UNDEF
    Не поддерживается
#IF ... #ENDIF
    Не поддерживается
#include
    Не поддерживается
$
    Подобно InStr()
%
    Mod
& (макроподстановка)
```

Не поддерживается :: (выполнение метода в родительском классе)
 Не поддерживается
 ? | ??
 Похожо *Print*
 ???
 Не поддерживается
 ABS()
 Abs()
 ACLASS()
 Не поддерживается
 ACOPY()
 Не поддерживается
 ACOS()
 Не поддерживается
 ADATABASES()
 Не поддерживается
 ADBOBJECTS()
 Не поддерживается
 ADD CLASS
 Не поддерживается
 ADD TABLE
 Не поддерживается
 ADEL()
 Не поддерживается
 ADIR()
 Не поддерживается
 AELEMENT()
 Не поддерживается
 AERROR()
 Похожо коллекции объектов Err в DAO
 AFIELDS()
 Не поддерживается
 AFONT()
 Не поддерживается
 AINS()
 Не поддерживается
 AINSTANCE()
 Не поддерживается
 ALEN()
 Не поддерживается
 ALLTRIM()
 Trim()
 ALTER TABLE - SQL
 Не поддерживается
 AMEMBERS()
 Не поддерживается
 APPEND
 Метод AddNew объекта RecordSet в DAO
 APPEND FROM
 Не поддерживается
 APPEND FROM ARRAY
 Не поддерживается
 APPEND MEMO
 Не поддерживается
 APPEND PROCEDURES
 Не поддерживается
 APRINTERS()
 Не поддерживается
 ASC()
 Asc()
 ASCAN()
 Не поддерживается
 ASELOBJ()
 Не поддерживается
 ASIN()
 Не поддерживается

ASORT()
 Не поддерживается
ASSERT
 Не поддерживается
ASUBSCRIPT()
 Не поддерживается
AT()
 Подобно **InStr()**. Нет поддержки числа вхождений выражения для поиска
ATAN()
 Не поддерживается
ATC()
 Подобно **InStr()**. Нет поддержки числа вхождений выражения для поиска
ATCC()
 Не поддерживается
ATCLINE()
 Не поддерживается
ATLINE()
 Не поддерживается
ATN2()
 Не поддерживается
AT_C()
 Не поддерживается
AUSED()
 Не поддерживается
AVERAGE
 Не поддерживается
BEGIN TRANSACTION
 BeginTrans
BETWEEN()
 Не поддерживается
BINTOC()
 Не поддерживается
BITAND()
 Не поддерживается
BITCLEAR()
 Не поддерживается
BITLSHIFT()
 Не поддерживается
BITNOT()
 Не поддерживается
BITOR()
 Не поддерживается
BITRSHIFT()
 Не поддерживается
BITSET()
 Не поддерживается
BITTEST()
 Не поддерживается
BITXOR()
 Не поддерживается
BLANK
 Не поддерживается
BOF()
 Свойство BOF объект RecordSet в DAO
CALCULATE
 Не поддерживается
CANCEL
 End
CANDIDATE()
 Не поддерживается
CAPSLOCK()
 Не поддерживается
CD | CHDIR
 ChDir
CDOW()
 Не поддерживается

CDX()
 Не поддерживается
 CEILING()
 Не поддерживается
 CHR()
 Chr()
 CHRSAW()
 Не поддерживается
 CHRTRAN()
 Не поддерживается
 CHRTRANC()
 Не поддерживается
 CLEAR RESOURCES
 Не поддерживается
 CLOSE
 Не поддерживается
 CLOSE INDEXES
 Не поддерживается
 CLOSE TABLES
 Не поддерживается
 CMONTH()
 Не поддерживается
 COMPILE DATABASE
 Не поддерживается
 COMPOBJ()
 Не поддерживается
 CONTINUE
 Не поддерживается
 COPY FILE
 Не поддерживается
 COPY INDEXES
 Не поддерживается
 COPY MEMO
 Не поддерживается
 COPY PROCEDURES
 Не поддерживается
 COPY STRUCTURE
 Не поддерживается
 COPY STRUCTURE EXTENDED
 Не поддерживается
 COPY TAG
 Не поддерживается
 COPY TO
 Не поддерживается
 COPY TO ARRAY
 Не поддерживается
 COS()
 Cos()
 COUNT
 Не поддерживается
 CPCONVERT()
 Не поддерживается
 CPCURRENT()
 Не поддерживается
 CPDBF()
 Не поддерживается
 CREATE
 Не поддерживается
 CREATE CLASS
 Не поддерживается
 CREATE CLASSLIB
 Не поддерживается
 CREATE CONNECTION
 Подобно свойству Connect в DAO
 CREATE DATABASE
 Метод CreateDatabase объекта Work-space в DAO

CREATE FROM
 Не поддерживается
CREATE QUERY
 Метод CreateQueryDef объекта DataBase в DAO
CREATE SQL VIEW
 Подобно свойству SQL объекта QueryDef в DAO
CREATE TABLE - SQL
 Метод Execute объекта Database в DAO
CREATE TRIGGER
 Не поддерживается
CREATEBINARY()
 Не поддерживается
CREATEOBJECT()
 CreateObject()
CREATEOFFLINE()
 Не поддерживается
CTOBIN()
 Не поддерживается
CTOD()
 CDate()
CTOT()
 Не поддерживается
CURDIR()
 CurDir()
CURSORGETPROP()
 Не поддерживается
CURSORSETPROP()
 Не поддерживается
DATE()
 Date()
DATETIME()
 Не поддерживается
DAY()
 Day()
DBC()
 Не поддерживается
DBGETPROP()
 Не поддерживается
DBSETPROP()
 Не поддерживается
DBUSED()
 Не поддерживается
DEBUG
 Не поддерживается
DEBUGOUT
 Не поддерживается
DECLARE
Dim и ReDim
DECLARE - DLL
 Declare
DEFINE CLASS
 Не поддерживается
DELETE
 Подобно методу Delete объекта RecordSet в DAO
DELETE - SQL
 Метод Execute объекта Database в DAO
DELETE CONNECTION
 Не поддерживается
DELETE DATABASE
 Не поддерживается
DELETE FILE
 Kill
DELETE TAG
 Не поддерживается
DELETE TRIGGER
 Не поддерживается

DELETE VIEW
 Не поддерживается
DELETED()
 Не поддерживается
DESCENDING()
 Не поддерживается
DIFFERENCE()
 Подобно *Like*
DIMENSION
Dim и *ReDim*
DIR or DIRECTORY
 Не поддерживается
DIRECTORY()
 Подобно *Dir()*
DISKSPACE()
 Не поддерживается
DISPLAY
 Не поддерживается
DISPLAY CONNECTIONS
 Не поддерживается
DISPLAY DATABASE
 Не поддерживается
DISPLAY DLLS
 Не поддерживается
DISPLAY FILES
 Не поддерживается
DISPLAY MEMORY
 Не поддерживается
DISPLAY OBJECTS
 Не поддерживается
DISPLAY PROCEDURES
 Не поддерживается
DISPLAY STATUS
 Не поддерживается
DISPLAY STRUCTURE
 Не поддерживается
DISPLAY TABLES
 Не поддерживается
DISPLAY VIEWS
 Не поддерживается
DMY()
 Не поддерживается
DO
 Call
DO CASE ... ENDCASE
 Select Case... End
DO WHILE ... ENDDO
 Do While... End
DODEFAULT()
 Не поддерживается
DOEVENTS
 DoEvents
DOW()
 WeekDay()
DROPOFFLINE()
 Не поддерживается
DTOC()
 Str()
DTOR()
 Не поддерживается
DTOS()
 Не поддерживается
DTOT()
 Не поддерживается
DefaultExt()
 Не поддерживается

DriveType()
 Не поддерживается
 EJECT
 Не поддерживается
 EJECT PAGE
 Не поддерживается
 EMPTY()
 Подобно *IsEmpty()*. Проверяется только инициализация
 END TRANSACTION
 CommitTrans
 EOF()
 Свойство EOF объекта RecordSet в DAO
 ERASE
 Kill
 ERROR
 Подобно Error
 ERROR()
 Подобно свойству Number объекта Err в DAO
 EVALUATE()
 Не поддерживается
 EXIT
 Exit
 EXP()
 Exp()
 EXPORT
 Не поддерживается
 FCHSIZE()
 Не поддерживается
 FCLOSE()
 Close
 FCOUNT()
 Не поддерживается
 FCREATE()
 Подобно Open. Не поддерживается указание типа файла
 FDATE()
 Подобно *FileDateTime()*
 FEOF()
 EOF()
 FERROR()
 Не поддерживается
 FFLUSH()
 Не поддерживается
 FGETS()
 Подобно *Input #*
 FIELD()
 Не поддерживается
 FILE()
 Подобно *Dir()*
 FILTER()
 Не поддерживается
 FKLABEL()
 Не поддерживается
 FKMAX()
 Не поддерживается
 FLOCK()
 Не поддерживается
 FLOOR()
 Не поддерживается
 FLUSH
 Не поддерживается
 FONTMETRIC()
 Не поддерживается
 FOPEN()
 Open
 FOR ... NEXT
 For... Next

FOR()
 Не поддерживается
 FOUND()
 Не поддерживается
 FPUTS()
 Подобно *Print #*
 FREAD()
 Подобно *Input #*
 FREE TABLE
 Не поддерживается
 FSEEK()
 Seek()
 FSIZE()
 Не поддерживается
 FTIME()
 Подобно *FileDateTime()*
 FULLPATH()
 Не поддерживается
 FUNCTION
 Function
 FV()
 FV()
 FWRITE()
 Подобно *Print #*
 GATHER
 Не поддерживается
 GETCOLOR()
 Метод ShowColor элемента ActiveX CommonDialog
 GETCP()
 Не поддерживается
 GETDIR()
 Не поддерживается
 GETENV()
 Environ()
 GETEXPR
 Не поддерживается
 GETFILE()
 Метод ShowOpen элемента ActiveX CommonDialog
 GETFLDSTATE()
 Не поддерживается
 GETFONT()
 Метод ShowFont элемента ActiveX CommonDialog
 GETNEXTMODIFIED()
 Не поддерживается
 GETOBJECT()
 GetObject()
 GETPEM()
 Не поддерживается
 GETPICT()
 Не поддерживается
 GETPRINTER()
 Подобно методу ShowPrint элемента ActiveX CommonDialog
 GO | GOTO
 Подобно методу Move объекта RecordSet в DAO
 GOMONTH()
 Не поддерживается
 HEADER()
 Не поддерживается
 HELP
 Метод ShowHelp элемента ActiveX CommonDialog
 HOME()
 Не поддерживается
 HOUR()
 Hour()
 IDXCOLLATE()
 Не поддерживается

IF ... ENDIF
 If Then... Endif
 IIF()
 IIf()
 IMESTATUS()
 Не поддерживается
 IMPORT
 Не поддерживается
 INDBC()
 Не поддерживается
 INDEX
 Не поддерживается
 INKEY()
 Не поддерживается
 INLIST()
 Не поддерживается
 INSERT - SQL
 Метод Execute объекта Recordset в DAO
 INSMODE()
 Не поддерживается
 INT()
 Int()
 ISALPHA()
 Не поддерживается
 ISBLANK()
 Не поддерживается
 ISDIGIT()
 IsNumeric()
 ISEXCLUSIVE()
 Не поддерживается
 ISFLOCKED()
 Не поддерживается
 ISLEADBYTE()
 Не поддерживается
 ISLOWER()
 Конструкция $g = LCase(g)$
 ISNULL()
 IsNull()
 ISREADONLY()
 Не поддерживается
 ISRLOCKED()
 Не поддерживается
 ISUPPER()
 Конструкция $g = LCase(g)$
 KEY()
 Не поддерживается
 KEYBOARD
 Не поддерживается
 KEYMATCH()
 Не поддерживается
 LASTKEY()
 Не поддерживается
 LEFT()
 Left()
 LEFTC()
 Не поддерживается
 LEN()
 Len()
 LENC()
 Не поддерживается
 LIKE()
 Like
 LIKEC()
 Не поддерживается
 LINENO()
 Не поддерживается

LOADPICTURE()
 LoadPicture()
 LOCAL
 Private
 LOCATE
 Не поддерживается
 LOCFILE()
 Не поддерживается
 LOCK()
 Не поддерживается
 LOG()
 Log()
 LOG10()
 Не поддерживается
 LOOKUP()
 Не поддерживается
 LOWER()
 LCase()
 LPARAMETERS
 Не поддерживается
 LTRIM()
 LTrim()
 LUPDATE()
 Не поддерживается
 MAX()
 Не поддерживается
 MCOL()
 Не поддерживается
 MD | MKDIR
 MdDir
 MDOWN()
 Не поддерживается
 MDX()
 Не поддерживается
 MDY()
 Не поддерживается
 MEMLINES()
 Не поддерживается
 MESSAGE()
 Error()
 MESSAGEBOX()
 MsgBox()
 MIN()
 Не поддерживается
 MINUTE()
 Minute()
 MLINE()
 Не поддерживается
 MOD()
 Mod
 MODIFY CLASS
 Не поддерживается
 MODIFY COMMAND
 Не поддерживается
 MODIFY CONNECTION
 Не поддерживается
 MODIFY DATABASE
 Не поддерживается
 MODIFY FILE
 Не поддерживается
 MODIFY GENERAL
 Не поддерживается
 MODIFY MEMO
 Не поддерживается
 MODIFY QUERY
 Не поддерживается

MODIFY STRUCTURE
 Не поддерживается
MODIFY VIEW
 Не поддерживается
MONTH()
 Month()
MOUSE
 Не поддерживается
MROW()
 Не поддерживается
MTON()
 Val()
NDX()
 Не поддерживается
NODEFAULT
 Не поддерживается
NORMALIZE()
 Не поддерживается
NOTE
 Rem
NTOM()
 CCur()
NUMLOCK()
 Не поддерживается
NVL()
 Не поддерживается
OBJTOCLIENT()
 Не поддерживается
OCCURS()
 Не поддерживается
OLDVAL()
 Не поддерживается
ON ERROR
 On Error
ON KEY LABEL
 Не поддерживается
ON()
 Не поддерживается
OPEN DATABASE
 Не поддерживается
ORDER()
 Не поддерживается
OS()
 Не поддерживается
PACK
 Не поддерживается
PACK DATABASE
 Не поддерживается
PADC()
 Не поддерживается
PADL()
 Не поддерживается
PADR()
 Не поддерживается
PARAMETERS
 Не поддерживается
PARAMETERS()
 Не поддерживается
PAYMENT()
 Pmt()
PCOL()
 Не поддерживается
PEMSTATUS()
 Не поддерживается
PI()
 Не поддерживается

POP KEY
Не поддерживается

PRIMARY()
Не поддерживается

PRINTSTATUS()
Не поддерживается

PRIVATE
Не поддерживается

PROCEDURE
Sub

PROGRAM()
Не поддерживается

PROPER()
Не поддерживается

PROW()
Не поддерживается

PRTINFO()
Не поддерживается

PUBLIC
Public

PUSH KEY
Не поддерживается

PUTFILE()
Метод ShowSave элемента ActiveX CommonDialog

PV()
PV()

RAND()
Rnd()

RAT()
Не поддерживается

RATC()
Не поддерживается

RATLINE()
Не поддерживается

RD | RMDIR
RmDir

RECALL
Не поддерживается

REFRESH()
Не поддерживается

REINDEX
Не поддерживается

RELATION()
Не поддерживается

RELEASE
Не поддерживается

RELEASE CLASSLIB
Не поддерживается

REMOVE CLASS
Не поддерживается

REMOVE TABLE
Метод Delete коллекции TableDefs

RENAME
Name

RENAME CLASS
Не поддерживается

RENAME CONNECTION
Не поддерживается

RENAME TABLE
Не поддерживается

RENAME VIEW
Не поддерживается

REPLACE
Не поддерживается

REPLACE FROM ARRAY
Не поддерживается

REPLICATE()
 String()
 REQUERY()
 Не поддерживается
 RESTORE FROM
 Не поддерживается
 RESUME
 Resume
 RETRY
 Не поддерживается
 RETURN
 Подобно Return. Не все опции поддерживаются
 RGB()
 Не поддерживается
 RIGHT()
 Right()
 RIGHTC()
 Не поддерживается
 RLOCK()
 Не поддерживается
 ROLLBACK
 Rollback
 ROUND()
 Не поддерживается
 RTOD()
 Не поддерживается
 RTRIM()
 RTrim()
 RUN | !
 Shell()
 SAVE TO
 Не поддерживается
 SAVEPICTURE()
 SavePicture()
 SCAN ... ENDSCAN
 Не поддерживается
 SCATTER
 Не поддерживается
 SEC()
 Second()
 SECONDS()
 Timer()
 SEEK
 Метод Seek объекта RecordSet в DAO
 SEEK()
 Не поддерживается
 SELECT
 Не поддерживается
 SELECT - SQL
 Метод Execute объекта RecordSet в DAO
 SET ANSI
 Не поддерживается
 SET AUTOSAVE
 Не поддерживается
 SET BLOCKSIZE
 Не поддерживается
 SET CENTURY
 Не поддерживается
 SET CLASSLIB
 Не поддерживается
 SET COLLATE
 Не поддерживается
 SET CPDIALOG
 Не поддерживается
 SET CURRENCY
 Не поддерживается

SET DATABASE
 Не поддерживается
SET DATASESSION
 Не поддерживается
SET DATE
 Не поддерживается
SET DELETED
 Не поддерживается
SET EXACT
 Не поддерживается
SET FDOW
 Не поддерживается
SET FIELDS
 Не поддерживается
SET FILTER
 Не поддерживается
SET
 Не поддерживается
SET FWEEK
 Не поддерживается
SET HELP
 Не поддерживается
SET INDEX
 Не поддерживается
SET KEY
 Не поддерживается
SET LOGERRORS
 Не поддерживается
SET MEMOWIDTH
 Не поддерживается
SET NEAR
 Не поддерживается
SET NOCPTRANS
 Не поддерживается
SET NULL
 Не поддерживается
SET OLEOBJECT
 Не поддерживается
SET OPTIMIZE
 Не поддерживается
SET ORDER
 Не поддерживается
SET PATH
 Не поддерживается
SET RELATION
 Подобно методу CreateField объекта Relation в DAO
SET RELATION OFF
 Метод Delete объекта Relation
SET SKIP
 Не поддерживается
SET SYSFORMATS
 Не поддерживается
SET TRBETWEEN
 Не поддерживается
SET TYPEAHEAD
 Не поддерживается
SET UNIQUE
 Не поддерживается
SET()
 Не поддерживается
SETFLDSTATE()
 Не поддерживается
SIGN()
 Sgn()
SIN()
 Sin()

SKIP

Методы MovePrevious и MoveNext объекта Recordset в DAO

SORT

Не поддерживается

SOUNDEX()

Не поддерживается

SPACE()

Space()

SQLCANCEL()

Не поддерживается

SQLCOLUMNS()

Не поддерживается

SQLCOMMIT()

Не поддерживается

SQLCONNECT()

Не поддерживается

SQLDISCONNECT()

Не поддерживается

SQLEXEC()

Не поддерживается

SQLGETPROP()

Не поддерживается

SQLMORERESULTS()

Не поддерживается

SQLPREPARE()

Не поддерживается

SQLROLLBACK()

Не поддерживается

SQLSETPROP()

Не поддерживается

SQLSTRINGCONNECT()

Не поддерживается

SQLTABLES()

Не поддерживается

SQRT()

Sqr()

STORE

Не поддерживается

STR()Подобно *Str()* и *Format()***STRCONV()**

Не поддерживается

STRTRAN()

Не поддерживается

STUFF()

Подобно Mid

STUFFC()

Не поддерживается

SUBSTR()

Mid()

SUBSTRC()

Не поддерживается

SUM

Не поддерживается

SUSPEND

Не поддерживается

SYS(1)

Не поддерживается

SYS(10)

Не поддерживается

SYS(1037)

Не поддерживается

SYS(11)

Не поддерживается

SYS(1269)

Не поддерживается

SYS(1270)
 Не поддерживается
 SYS(1271)
 Не поддерживается
 SYS(1272)
 Не поддерживается
 SYS(13)
 Не поддерживается
 SYS(14)
 Не поддерживается
 SYS(16)
 Не поддерживается
 SYS(17)
 Не поддерживается
 SYS(2000)
 Не поддерживается
 SYS(2001)
 Не поддерживается
 SYS(2002)
 Не поддерживается
 SYS(2007)
 Не поддерживается
 SYS(2014)
 Не поддерживается
 SYS(2015)
 Не поддерживается
 SYS(2018)
 Не поддерживается
 SYS(2020)
 Не поддерживается
 SYS(2021)
 Не поддерживается
 SYS(2022)
 Не поддерживается
 SYS(2029)
 Не поддерживается
 SYS(21)
 Не поддерживается
 SYS(3)
 Не поддерживается
 SYS(3051)
 Не поддерживается
 SYS(3052)
 Не поддерживается
 SYS(3053)
 Не поддерживается
 SYS(5)
 CurDir() при использовании LEFT(CURDIR(),2)
 SYS(6)
 Не поддерживается
 SYS(9)
 Не поддерживается
 SetClipDat()
 Не поддерживается
 StrFilter()
 Не поддерживается
 TABLEREVERT()
 Не поддерживается
 TABLEUPDATE()
 Не поддерживается
 TAG()
 Не поддерживается
 TAN()
 Tan()
 TARGET()
 Не поддерживается

TIME()
 Подобно *Time()*. Имеет меньшую точность
TOTAL
 Не поддерживается
TRANSFORM()
 Подобно *Format()*
TRIM()
RTrim()
TTOC()
 Не поддерживается
TTOD()
 Не поддерживается
TXNLEVEL()
 Не поддерживается
TXTWIDTH()
 Не поддерживается
TYPE
 Не поддерживается
TYPE()
TypeName()
UNLOCK
 Не поддерживается
UPDATE - SQL
 Не поддерживается
UPPER()
UCase()
USE
 Не поддерживается
USED()
 Не поддерживается
VAL()
Val()
VALIDATE DATABASE
 Не поддерживается
VERSION()
 Не поддерживается
WAIT
 Не поддерживается
WEEK()
 Не поддерживается
WEXIST()
 Не поддерживается
WITH ... ENDWITH
With... End With
WLAST()
 Не поддерживается
WLCOL()
 Не поддерживается
WLROW()
 Не поддерживается
WMAXIMUM()
 Не поддерживается
WMINIMUM()
 Не поддерживается
WONTOP()
 Не поддерживается
WOUTPUT()
 Не поддерживается
WVISIBLE()
 Не поддерживается
YEAR()
Year()
ZAP
 Не поддерживается
_TRIGGERLEVEL
 Не поддерживается