Федеральное агентство по образованию
Озёрский технологический институт (филиал)
ГОУ ВПО «Московский инженерно-физический институт
(государственный университет)»

Вл. Пономарев

Проектирование автоматизированных программных систем

Учебно-практическое пособие

Озёрск, 2011

УДК 681.3.06
П 56

Пономарев В.В. Проектирование автоматизированных программных систем. Учебно-практическое пособие. Редакция 10.10.2011. Озерск: ОТИ МИФИ, 2011. — 124 с., ил.

Учебное пособие является введением в технологию Microsoft OLE Automation, используемую для управления программными компонентами при помощи интерфейса диспетчеризации IDispatch. В пособии рассматриваются также различные аспекты создания настольных приложений, экспортирующих объектную модель через интерфейс IDispatch. Пособие предназначено для студентов, обучающихся по специальности 230105 — «Программное обеспечение вычислительной техники и автоматизированных систем».

Рецензенты:
1) Директор по ИТ ООО «ЛМТ», к.т.н., А.О. Ключев
2) Ст. преподаватель кафедры ПМ ОТИ МИФИ Д.Н. Бдехов

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ МИФИ

Содержание

Обозначения и сокращения

ActiveX — технологии, основанные на COM.

ActiveX EXE — сервер объектов COM типа out-of-process.

ADO — ActiveX Data Objects, объектная модель доступа к данным.

ADOX — ActiveX Data Objects Extensions, расширения объектной модели доступа к данным.

dpi — dot per pixel, точек на дюйм, единица измерения разрешающей способности точечного устройства вывода.

MS Excel — Microsoft Excel.

MS Word — Microsoft Word.

MSVC++ — Microsoft Visual C++ 6.0.

MSVB — Microsoft Visual Basic 6.0.

VBA — Visual Basic for Application.

Автоматизация — технология OLE Automation.

Диспинтерфейс — интерфейс диспетчеризации IDispatch.

Скрипт — программа на скриптовом языке программирования.

Введение

Дисциплина «Проектирование автоматизированных программных систем» изучает использование технологии *Microsoft OLE Automation* (далее для простоты называемую *автоматизацией*) для создания компонентов (серверов) *ActiveX EXE* и программного управления ими посредством других приложений или скриптовых языков (скриптов), называемых контроллерами автоматизации.

В основе технологии *OLE Automation* лежит базовая инфраструктура *Microsoft*, используемая для построения многократно используемых программных компонентов на основе *COM*-объектов. Данная инфраструктура должна быть изучена предварительно.

Главное отличие технологии *OLE Automation* заключается в наследовании интерфейса диспетчеризации *IDispatch* вместо интерфейса *IUnknown*. Интерфейс *IDispatch* наследуется от *IUnknown*, поэтому любой *COM*-объект, наследующий *IDispatch*, автоматически наследует и *IUnknown*, а потому является также полноценным *COM*-объектом.

Интерфейс диспетчеризации предназначен, в первую очередь, для управления *COM*-объектами из скриптовых языков программирования, а также из языков, не поддерживающих концепцию объектно-ориентированного программирования, таких, как *MSVB*. В этих языках отсутствуют средства для обращения к свойствам и методам объектов, а также средства создания самих объектов. Вместо этого в них встраиваются средства для создания указателя на интерфейс диспетчеризации и формирования вызовов его немногочисленных методов. В результате такие «простые» языки (называемые также «глупыми» в [1]) приобретают возможность «псевдообращения» к свойствам и методам объекта автоматизации.

В курсе изучается три основных раздела:

1) основы технологии *OLE Automation*;

2) основы автоматизации *Microsoft Office*;

3) создание сервера автоматизации.

Изучение данного курса заканчивается *курсовым проектом*, в ходе которого разрабатывается приложение-сервер автоматизации.

В основе автоматизации лежит интерфейс диспетчеризации *IDispatch*. Интерфейс диспетчеризации позволяет стандартным способом использовать уникальную функциональность конкретного *COM*-объекта. Кроме того, автоматизация внедрила в операционную систему *Microsoft Windows* так называемые специальные типы автоматизации, такие, как *Variant*,

BSTR и *SafeArray*, а также расширенную информацию об ошибках, необязательные и поименованные аргументы процедур, и коллекции, имеющие большое значение в современном программировании.

Объектная модель автоматизации — это экспортируемая приложением иерархия классов (объектов *COM*), позволяющая полностью управлять приложением при помощи контроллера автоматизации. Следует отметить, что объектная модель приложения позволяет выполнять все действия, которые может выполнить пользователь приложения при помощи графического интерфейса, вплоть до управления линейками прокрутки и окнами приложения или доступа к любой, самой маленькой, части данных, которыми приложение управляет. Наиболее «продвинутыми» в смысле автоматизации являются приложения *Microsoft Office*, которые оснащены встроенным языком программирования *VBA*. Примеру *Microsoft* последовали и некоторые другие известные производители программных продуктов, такие, как *Autodesk*. Приложение *AutoCAD*, начиная с версии 15, также является полностью автоматизированным приложением, что позволяет использовать его для создания графических приложений специального назначения или для выполнения графических операций в автоматическом (в смысле автоматизации) режиме.

Автоматизация *Microsoft Office* и других приложений позволяет использовать их огромные потенциальные возможности в обработке информации определенного вида для быстрого создания приложений специального назначения или для решения практических задач автоматизации использования компьютера.

С одной стороны, на современном компьютере обычно установлено достаточно большое количество программ общего назначения. Так, *MS Word* — это универсальный программный компонент для обработки текстовой информации, включающей в себя разнородные объекты, такие, как рисунки, формулы, диаграммы и т.п. *MS Excel* — это универсальный вычислительный программный компонент, позволяющий с минимальными (нулевыми) затратами на программирование производить любые объёмы вычислений. Приложение *AutoCAD* — это универсальный программный компонент для решения любых видов графических задач.

Автоматизация этих компонентов позволяет использовать их вычислительные возможности в различных приложениях, предназначенных для решения часто возникающих конкретных практических задач. Так, например, *MS Word* может быть использован для проверки орфографии в программе для автоматического составления писем, *MS Excel* — для построе-

ния диаграммы в программе для анализа результатов эксперимента, а *AutoCAD* — для построения чертежа выкройки в программе для моделирования одежды.

Сервер автоматизации — это исполняемый файл *ActiveX EXE*, экспортирующий объектную модель. Мы различаем серверы *ActiveX EXE* по способу создания объекта автоматизации. Если объект автоматизации создается только контроллером автоматизации, сервер этого объекта выступает только как сервер автоматизации. Примером может служить сервер, выполняющий функции музыкального центра приложения.

С другой стороны, сервер может выполнять функции как обычного приложения, так и сервера автоматизации. Это наиболее сложный тип приложения, позволяющий использовать приложение одновременно для работы с пользователем и для *программного управления*, при помощи, например, контроллера автоматизации или скрипта. Создание приложения, экспортирующего объектную модель, является весьма непростой задачей. Прежде всего, объектная модель — результат кропотливого труда по разработке классов, описывающих предметную область. Это предполагает детальный анализ структур данных, с которыми работает приложение, с целью выделить в них объекты автоматизации и описать их поведение и взаимоотношения. Эта работа окупает себя, так как созданная модель позволяет легко управлять структурами данных не только во время автоматизации, но и во время работы приложения как приложения. Иначе говоря, объектная модель создается не столько для автоматизации, сколько для упорядочения управления приложением как таковым. Само приложение в этом случае использует свою объектную модель для работы в обычном (вообще-то в любом) режиме. Классическими примерами такой модели поведения приложений являются приложения *Microsoft Office*.

Следует также отметить, что изучаемая технология является особенностью операционной системы *Microsoft Windows*. Тем не менее, принципы построения приложений автоматизации очень полезны, так как позволяют упорядочить объектные модели в приложениях, создаваемых в других средах.

Основы автоматизации

Терминология

Интерфейс — совокупность методов и свойств, экспортируемых классом *COM*-объекта. С технической точки зрения и методы и свойства *COM*-объекта являются функциями. Разделение функций интерфейса на методы и свойства основано на способе их использования. Свойство может быть использовано в операторе присваивания или в выражении. Если свойство используется в левой части оператора присваивания, то говорят, что свойство устанавливается (записывается его новое значение). Во всех других случаях говорят, что свойство читается. Методы используются обычным для функций способом (вызовом). В отличие от свойств, методы обычно имеют параметры (аргументы). Заметим, что свойства также могут иметь параметры (индексы), однако это не является для них характерным.

Интерфейс *COM* является объявляемым программным компонентом, не имеющим кода (абстрактным классом, состоящим только из чистых виртуальных функций). Это означает, что сначала описывается интерфейс *COM*-объекта, а затем создается сам *COM*-объект. Для описания интерфейсов используется язык *IDL* (*Interface Definition Language*).

Интерфейсы *COM* принято разделять на стандартные и пользовательские. *Стандартный* интерфейс *COM* — это базовый интерфейс, обеспечивающий функционирование *COM*-технологии. К стандартным относят интерфейсы *IUnknown*, *IDispatch*, *IClassFactory* и множество других. Собственный интерфейс объекта *COM*, описывающий его уникальную функциональность, называют *пользовательским* (*custom*) интерфейсом.

Объектная модель — это экспортируемая сервером *ActiveX EXE* иерархия классов *COM*, в которой отдельные классы отражают реальные элементы структур данных приложения и их взаимоотношения. Классы объектной модели связаны между собой взаимными ссылками, позволяющими выполнять переходы из одного класса в другой или обращаться к свойствам и методам одних классов из других. Подробнее объектная модель рассматривается в разделе «Создание сервера автоматизации».

Сервер *ActiveX EXE*, экспортирующий один или несколько классов *COM*, наследующих интерфейс *IDispatch*, и образующих объектную модель, называют *сервером автоматизации*. Сервер должен быть зарегистрирован в реестре *Windows*.

Объект *COM*, экспортируемый сервером автоматизации, будем называть *объектом автоматизации*. Как правило, объектом автоматизации является верхний класс иерархии объектной модели приложения. Общепринято называть этот класс *Application* (приложение).

COM класс (кокласс) объекта автоматизации будем называть *классом автоматизации*.

Приложение, управляющее объектом автоматизации *через интерфейс диспетчеризации*, называется *контроллером автоматизации*. В качестве контроллера автоматизации могут выступать как обычные приложения *Windows*, так и скрипты, в том числе скрипты на страницах *HTML*.

IDispatch

В основе технологии *OLE Automation* лежит интерфейс *IDispatch*, иначе называемый *интерфейсом диспетчеризации* или *диспинтерфейсом*.

Диспинтерфейс управляет любым объектом автоматизации одинаковым, стандартным способом — вызовом одних и тех же методов интерфейса диспетчеризации. В этой ситуации возникает противоречие, заключающееся в том, что конкретный класс автоматизации обладает собственным (пользовательским) интерфейсом, описываемым уникальным набором функций, а их вызов производится одинаково для любого пользовательского интерфейса — при помощи метода **`IDispatch::Invoke`**.

Изюминкой диспинтерфейса (или волшебством, *magic*, как сказано в [2]) является перенаправление вызовов метода **`Invoke`** на необходимые методы и свойства пользовательского интерфейса. Перенаправление, или диспетчеризация, дало повод для названия интерфейса.

В основе диспетчеризации лежит *dispid* — номер, присваиваемый каждому методу или свойству пользовательского интерфейса. При вызове метода **`IDispatch::Invoke`** этот номер определяет, куда следует перенаправить вызов (рисунок 1).

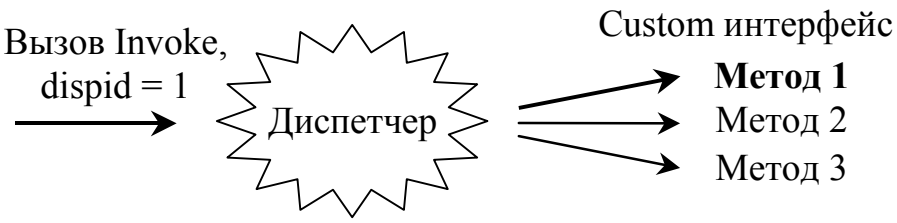

Рисунок 1 — Диспетчеризация вызовов

Номера методам и свойствам диспинтерфейса присваиваются достаточно произвольно, следует только помнить, что номер должен быть положительным числом. Среда *Visual Studio* проставляет номера автоматически, в порядке их объявления, начиная с единицы. Номер «ноль» и отрицательные номера используются для указания на специальные функции пользовательского интерфейса.

Диспетчер вызовов является частью диспинтерфейса и создается во время проектирования класса автоматизации, поэтому интерфейс диспетчеризации конкретного класса автоматизации является уникальным, или, говоря иначе, — *все интерфейсы диспетчеризации являются разными*.

Методы диспинтерфейса

Описание диспинтерфейса можно найти в файле *oaidl.idl* (приведено со значительными сокращениями):

```
interface IDispatch : IUnknown {

    HRESULT GetTypeInfoCount(
        [out]  UINT * pctinfo
    );
    HRESULT GetTypeInfo(
        [in]  UINT iTInfo,
        [in]  LCID lcid,
        [out] ITypeInfo ** ppTInfo
    );
    HRESULT GetIDsOfNames(
        [in]  REFIID riid,
        [in]  LPOLESTR * rgszNames,
        [in]  UINT cNames,
        [in]  LCID lcid,
        [out] DISPID * rgDispId
    );
    HRESULT Invoke(
        [in]  DISPID dispIdMember,
        [in]  REFIID riid,
        [in]  LCID lcid,
        [in]  WORD wFlags,
        [in, out] DISPPARAMS * pDispParams,
        [out] VARIANT * pVarResult,
        [out] EXCEPINFO * pExcepInfo,
        [out] UINT * puArgErr
    );
};
```

В диспинтерфейсе 4 метода: два первых используются для извлечения информации о типах, и два оставшихся — для собственно вызова методов пользовательского интерфейса.

Метод **`GetTypeInfoCount`** возвращает признак, указывающий на наличие информации о типах.

Метод **`GetTypeInfo`** предназначен для извлечения информации о типах, если она есть. Подробно эти методы здесь не рассматриваются.

Метод **`GetIDsOfNames`** позволяет получить *dispid* конкретного наименования метода или свойства. Параметры метода:

`riid` — зарезервировано, всегда **`IID_NULL`**;

`rgszNames` — указатель на массив имён запрашиваемых методов;

`cNames` — количество запрашиваемых имён;

`lcid` — идентификатор локали;

`rgDispId` — возвращаемый массив *dispid*.

Работа автоматизации основана на строковых значениях — названиях методов, свойств, классов и серверов. Когда пользователь запрашивает идентификатор того или иного метода, или создает объект автоматизации (см. ниже), он указывает строковый параметр-название. При этом результат запроса не обязательно будет положительным. В этом заключается весьма важная особенность автоматизации. Пользователь может только *предполагать*, что у объекта автоматизации есть метод с определенным наименованием, и запрашивает его идентификатор. В ответ он получает либо идентификатор, если наименование действительно существует, либо специальный идентификатор, указывающий на отсутствие наименования в данном диспинтерфейсе.

Другой важной особенностью автоматизации является возможность использования наименований, заданных на различных естественных языках. Например, диспинтерфейс можно построить таким образом, что он одинаково будет работать при использовании как английских, так и немецких, русских или других наименований. Поэтому при вызове методов диспинтерфейса следует указывать идентификатор локали, иначе говоря, — язык, на котором заданы наименования. Эта особенность должна быть заложена в диспинтерфейс при его проектировании. По умолчанию используемый язык только один — английский (американский).

Метод **`Invoke`** — главная «рабочая лошадка» диспинтерфейса. Он принимает идентификатор метода или свойства, а при необходимости и параметры метода или новое значение свойства, и формирует запрос к пользовательскому интерфейсу объекта автоматизации. После выполнения

запроса метод возвращает значение свойства или результат выполнения метода-функции, а также расширенную информацию об ошибке в случае ее возникновения или номер параметра, имеющего неправильный тип.

Параметры метода **Invoke**:

dispIdMember — *dispid* запрашиваемого метода или свойства;

riid — зарезервировано, всегда **IID_NULL**;

lcid — идентификатор локали;

wFlags — константа, указывающая на тип запроса.

Значениями могут быть:

DISPATCH_METHOD — запрашивается метод;

DISPATCH_PROPERTYGET — запрашивается чтение свойства;

DISPATCH_PROPERTYPUT — запрашивается запись свойства;

DISPID_PROPERTYPUTREF — запись свойства по ссылке;

pDispParams — параметры метода, индексы свойства, новое значение свойства при записи, упакованные в структуру **DISPPARAMS**. Подробнее см. ниже раздел «Упаковка параметров»;

pVarResult — результат, возвращаемый при чтении свойства или возвращаемое значение метода-функции;

pExcepInfo — расширенная информация об ошибке. Указывается идентификатор (номер) ошибки, её краткое описание, источник (наименование программного модуля), а также файл справки и идентификатор раздела справки, если файл справки задан;

puArgErr — номер первого параметра метода, имеющего неправильный тип;

На возникновение ошибки при выполнении метода, как и принято в *COM*, указывает ненулевое возвращаемое значение метода **Invoke**. Именно поэтому всегда следует вызывать метод **Invoke** как функцию, возвращающую **HRESULT**. Ненулевое возвращаемое значение является номером ошибки, описанным в файле *winerr.h*.

Упаковка параметров

При программировании задач автоматизации в среде *MSVC++* упаковка параметров является самой сложной задачей. Для упаковки используется структура **DISPPARAMS** (параметры диспетчеризации), описанная в файле *oaidl.idl* (приведено с незначительными сокращениями):

```
typedef struct tagDISPPARAMS {
    VARIANTARG * rgvarg;
    DISPID *     rgdispidNamedArgs;
```

```
    UINT          cArgs;
    UINT          cNamedArgs;
} DISPPARAMS;
```

Здесь:

rgvarg — массив структур **VARIANTARG**, содержащий параметры вызываемого метода или новое значение свойства при его записи. Структура **VARIANTARG** является полным аналогом структуры **VARIANT**, которая подробно описана ниже;

rgdispidNamedArgs — массив идентификаторов (*dispid*) поименованных параметров;

cArgs — общее количество параметров;

cNamedArgs — количество поименованных аргументов.

Прежде всего нужно определить, что называется поименованными параметрами. Во многих распространенных языках программирования общепринято использовать передачу параметров по их положению в определении процедуры. Такая передача параметров носит название *позиционной*.

Некоторые языки программирования, например, *MSVB*, и, соответственно, некоторые серверы автоматизации, могут принимать *поименованные параметры*. В автоматизации достаточно часто встречаются процедуры, принимающие более десяти параметров. Запомнить позицию каждого параметра такой процедуры достаточно сложно, поэтому вместо этого предлагается указывать параметры в произвольном порядке, но к каждому параметру при этом приписывать его наименование, так, как оно определено в исходном коде. Такой способ передачи параметров называется передачей поименованных параметров.

Например, если в языке *MSVB* определена процедура следующего вида:

```
Public Sub Foo(ByVal Arg_1, ByVal Arg_2, ByVal Arg_3, ByVal Arg_4),
```

то вызвать эту процедуру при разрешенных поименованных параметрах можно множеством различных способов. Вот некоторые из них:

```
Foo 10, 20, 30, 40
Foo 10, Arg_4:=40, Arg_2:=20, Arg_3:=30
Foo Arg_4:=40, Arg_2:=20, Arg_3:=30, Arg_1:=10
```

Здесь в первом примере используется только позиционная передача, во втором — смешанная, в третьем — только поименованная. Заметим, что

при смешанном способе передачи параметров сначала должны быть указаны все позиционные параметры, затем — поименованные.

Поскольку вызов метода в автоматизации разрешает использование поименованных параметров, возникает проблема их правильной передачи методу пользовательского интерфейса — на уровне машинного кода передача параметров производится только позиционно.

Для решения проблемы параметры метода упаковываются в структуру **DISPPARAMS** *в обратном порядке*, сначала позиционные, а затем поименованные. Чтобы определить правильную позицию поименованных параметров, создается массив идентификаторов поименованных параметров, в котором каждому поименованному параметру в том порядке, который образовался в массиве параметров, соответствует номер его позиции. Для второго примера передачи параметров, приведенного выше, массив параметров и массив идентификаторов содержат следующие значения:

Индекс	Массив параметров	Массив идентификаторов
0	30	2
1	20	1
2	40	3
3	10	

Как видим, массив идентификаторов короче массива параметров и соответствует по размеру количеству поименованных параметров. Последний параметр в массиве параметров — единственный позиционный. Упаковка параметров в обратном порядке помогает установить соответствие между значениями поименованных параметров и их позициями. Позиции параметров нумеруются, начиная с нуля.

На самом деле порядок поименованных параметров не имеет никакого значения, поскольку он в любом случае произвольный, и обратный порядок их перечисления принят для однообразия.

Кроме того, в автоматизации достаточно часто встречаются *необязательные* (*optional*) параметры. Необязательный параметр в массиве параметров должен указываться специальным образом. В поле **vt** структуры **VARIANTARG** записывается признак ошибки **VT_ERROR**, а в поле **scode** — значение **DISP_E_PARAMNOTFOUND** (параметр не найден).

Особый случай возникает при передаче нового значения свойства. При записи свойства требуется указать поименованный аргумент. В качестве имени аргумента используется константа **DISPID_PROPERTYPUT**, определенная в файле *oaidl.h*. Тогда, если передается только новое значение свойства, то указываются следующие значения в структуре **DISPPARAMS**:

```
cArgs = 1
cNamedArgs = 1
rgvarg[0].vt = тип_нового_значения
rgvarg[0].соответствующее_типу_имя = новое_значение
rgdispidNamedArgs[0] = DISPID_PROPERTYPUT
```

В заключение следует заметить, что если параметр метода передается по ссылке, то в поле **vt** структуры **VARIANTARG** должна быть добавлена константа **VT_BYREF**, а измененное значение параметра возвращается в ту же структуру. Во всех других случаях значения в этой структуре рассматриваются как константные.

Примеры вызовов метода Invoke

Ниже приводится комплексный пример использования функции **Invoke** для чтения и записи свойства **Visible**, а также для вызова метода **Evaluate** приложения *MS Excel*:

```
#include "stdafx.h"
#include <windows.h>
#include <objbase.h>
#include <oaidl.h>
void main() {
    OLECHAR * szPROGID = L"Excel.Application";
    OLECHAR * szVisible = L"Visible";
    OLECHAR * szEvaluate = L"Evaluate";
    LPDISPATCH pdisp = NULL;
    CLSID clsid;
    VARIANT pVarResult;
    DISPPARAMS dispparamsNoArgs = { NULL, NULL, 0, 0 };
    DISPPARAMS dispparams;
    DISPID dispid;
    DISPID dispidpp[1] = { DISPID_PROPERTYPUT };
    VARIANTARG vararg[1];
    HRESULT hr;
    VariantInit(&pVarResult);
    // Инициализируем OLE
    CoInitialize(0);
    // Получаем CLSID из ProgID
    hr = CLSIDFromProgID(szPROGID, &clsid);
    if (FAILED(hr)) goto error;
    // Создаем экземпляр COM-объекта, запрашиваем IDispatch
    hr = CoCreateInstance(clsid, NULL, CLSCTX_SERVER,
        IID_IDispatch, (void**)&pdisp);
    if (FAILED(hr)) goto error;
```

```
    // ЧИТАЕМ СВОЙСТВО Visible
    // Получаем dispid свойства Visible (равный 558)
    hr = pdisp->GetIDsOfNames(IID_NULL, &szVisible, 1,
        LOCALE_USER_DEFAULT, &dispid);
    if (FAILED(hr)) goto error;
    // Метод Invoke
    hr = pdisp->Invoke(dispid, IID_NULL, LOCALE_USER_DEFAULT,
        DISPATCH_PROPERTYGET, &dispparamsNoArgs, &pVarResult, 0, 0);
    if (FAILED(hr)) goto error;
    // Результат в pVarResult должен быть FALSE VT_BOOL
    // УСТАНАВЛИВАЕМ СВОЙСТВО Visible
    VariantInit(&vararg[0]);
    dispparams.rgvarg = vararg;
    dispparams.rgvarg[0].vt = VT_BOOL;
    dispparams.rgvarg[0].boolVal = -1;  // TRUE
    dispparams.cArgs = 1;
    dispparams.cNamedArgs = 1;
    dispparams.rgdispidNamedArgs = dispidpp;
    // Метод Invoke, dispid не изменился
    hr = pdisp->Invoke(dispid, IID_NULL, LOCALE_USER_DEFAULT,
        DISPATCH_PROPERTYPUT, &dispparams, 0, 0, 0);
    if (FAILED(hr)) goto error;
    // ВЫЗЫВАЕМ МЕТОД Evaluate с одним параметром
    // Получаем dispid метода Evaluate (равный 1)
    hr = pdisp->GetIDsOfNames(IID_NULL, &szEvaluate, 1,
        LOCALE_USER_DEFAULT, &dispid);
    if (FAILED(hr)) goto error;
    dispparams.rgvarg[0].vt = VT_BSTR;
    // ВЫЧИСЛЯЕМ ПРОИЗВЕДЕНИЕ 2*3
    dispparams.rgvarg[0].bstrVal = SysAllocString(L"2*3");
    dispparams.cArgs = 1;
    dispparams.cNamedArgs = 0;
    dispparams.rgdispidNamedArgs = NULL;
    // Метод Invoke для метода Evaluate
    hr = pdisp->Invoke(dispid, IID_NULL, LOCALE_USER_DEFAULT,
        DISPATCH_METHOD, &dispparams, &pVarResult, 0, 0);
    // Результат в pVarResult должен быть 6.0000000000000 VT_R8
error:
    // освобождаем объекты COM
    if (pdisp) pdisp->Release();
    // Деинициализируем OLE
    CoUninitialize();
}
```

Специальные dispid

Для некоторых категорий свойств и методов автоматизация определяет следующие специальные идентификаторы *dispid*:

DISPID_UNKNOWN (0) — возвращается методом **GetIDsOfNames** в случае, если запрашиваемое имя отсутствует в интерфейсе диспетчеризации;

DISPID_VALUE (-1) — указывает на свойство по умолчанию; для свойства по умолчанию разрешается опускать его наименование;

DISPID_PROPERTYPUT (-3) — указывает на наименование устанавливаемого свойства;

DISPID_NEWENUM (-4) — указывает на свойство **_NewEnum**, которое возвращает *объект перечисления*. Это ограниченное (*restricted*, невидимое) свойство. Используется в коллекциях;

DISPID_EVALUATE (-5) — указывает на метод **Evaluate**. Разрешает опускать наименование метода, заменяя его *квадратными скобками*. В следующем примере оба вызова идентичны:

```
x.[A1:C1].value = 10
x.Evaluate("A1:C1").value = 10
```

DISPID_CONSTRUCTOR (-6) — указывает на конструктор объекта;

DISPID_DESTRUCTOR (-7) — указывает на деструктор объекта;

DISPID_COLLECT (-8) — указывает на свойство **Collect**.

Зарезервированы также следующие *dispid*:

DISPID_Name — 800

DISPID_Delete — 801

DISPID_Object — 802

DISPID_Parent — 803

Кроме этого, *dispid* с номерами **500** и выше используются для обозначения свойств элементов управления *ActiveX*.

Специальные типы автоматизации

VARIANT

VARIANT — базовый тип автоматизации. Во многих скриптовых языках типы как таковые отсутствуют. Вместо обычных для других языков программирования типов используется только один — **VARIANT**.

VARIANT — это структура, которая может хранить значение любого типа, совместимого с автоматизацией (на практике часто говорят наоборот — любой тип, совместимый с **VARIANT**, или **VARIANT**-совместимый тип). Тип, совместимый с автоматизацией — на самом деле тип, который может сво-

бодно использоваться не только в автоматизации, но и вообще в программировании. При этом гарантируется его правильная интерпретация и преобразование в другой тип, совместимый с автоматизацией. Я бы рекомендовал всегда использовать только типы автоматизации.

Определение структуры приведено в файле *oaidl.idl*. Описание здесь сокращено и модифицировано. В комментариях указаны константы, назначаемые полю `vt` для указания типа хранимого значения:

```
struct __tagVARIANT {
    VARTYPE vt;
    WORD    wReserved1;
    WORD    wReserved2;
    WORD    wReserved3;
    union {
        LONG           lVal;        /* VT_I4                  */
        BYTE           bVal;        /* VT_UI1                 */
        SHORT          iVal;        /* VT_I2                  */
        FLOAT          fltVal;      /* VT_R4                  */
        DOUBLE         dblVal;      /* VT_R8                  */
        VARIANT_BOOL   boolVal;     /* VT_BOOL                */
        SCODE          scode;       /* VT_ERROR               */
        CY             cyVal;       /* VT_CY                  */
        DATE           date;        /* VT_DATE                */
        BSTR           bstrVal;     /* VT_BSTR                */
        IUnknown *     punkVal;     /* VT_UNKNOWN             */
        IDispatch *    pdispVal;    /* VT_DISPATCH            */
        SAFEARRAY *    parray;      /* VT_ARRAY               */
        /* Указательные типы                                  */
        BYTE *         pbVal;       /* VT_BYREF | VT_UI1      */
        SHORT *        piVal;       /* VT_BYREF | VT_I2       */
        LONG *         plVal;       /* VT_BYREF | VT_I4       */
        FLOAT *        pfltVal;     /* VT_BYREF | VT_R4       */
        DOUBLE *       pdblVal;     /* VT_BYREF | VT_R8       */
        VARIANT_BOOL * pboolVal;    /* VT_BYREF | VT_BOOL     */
        SCODE *        pscode;      /* VT_BYREF | VT_ERROR    */
        CY *           pcyVal;      /* VT_BYREF | VT_CY       */
        DATE *         pdate;       /* VT_BYREF | VT_DATE     */
        BSTR *         pbstrVal;    /* VT_BYREF | VT_BSTR     */
        IUnknown **    ppunkVal;    /* VT_BYREF | VT_UNKNOWN  */
        IDispatch **   ppdispVal;   /* VT_BYREF | VT_DISPATCH */
        SAFEARRAY **   pparray;     /* VT_BYREF | VT_ARRAY    */
        VARIANT *      pvarVal;     /* VT_BYREF | VT_VARIANT  */
        PVOID          byref;       /* VT_BYREF               */
    }
};
```

Структура **VARIANT** имеет размер 16 байт. Значения, размер которых не превышает 8 байт, записываются непосредственно в структуру. К ним относятся все числовые значения. Значения, размер которых превышает 8 байт, записываются в динамическую память, а в структуру **VARIANT** записывается указатель (для объектов, строк и массивов).

Поскольку **VARIANT** может хранить значения разных типов, первый элемент структуры **vt** является указателем типа значения.

Кроме указанных в комментариях констант, указывающих на тип хранимого значения, используются еще две:

VT_EMPTY — указывает на отсутствие значения (пусто);

VT_NULL — указывает на значение **NULL**, возвращаемое в запросах к базам данных при помощи языка *SQL*.

В приведенной выше структуре указаны только типы, совместимые с автоматизацией. К ним относятся следующие (наименования типов соответствуют языку *MSVB*):

Boolean — логическое значение **False** (**0**) или **True** (**-1**);

Byte — беззнаковое целое число (1 байт);

Integer — знаковое целое число (2 байта);

Long — знаковое целое число (4 байта);

Single — вещественное число одинарной точности (4 байта);

Double — вещественное число двойной точности (8 байт);

Date — дата, включающая в себя время, или время (8 байт);

Currency — точное вещественное число (8 байт);

String — строковое значение (длина строки до 2 Гбайт символов);

Variant — значение типа **VARIANT**;

Object — объект (указатель на интерфейс **IUnknown** или **IDispatch**);

Кроме этого, структура может хранить указатель (ссылку) на любой из перечисленных типов, массивы этих типов, и собственно массив.

В случае, если поле **vt** содержит значение **VT_ERROR**, структура содержит код ошибки *SCODE*.

Тип **Currency** — это точное вещественное число, имеющее 19-20 значащих цифр. Количество знаков после запятой фиксировано и равно четырем.

Для работы с типом **VARIANT** следует использовать системные функции. Перечень и описание функций приведены в приложении А.

Порядок использования типа **VARIANT** следующий:

1) инициализировать (**VariantInit**);

2) записать тип данных в поле **vt**;

3) записать значение в соответствующее типу поле объединения;

4) выполнить операции;

5) очистить переменную (**VariantClear**).

Следующий пример поясняет сказанное:

```
VARIANT V;
VariantInit(&V);
V.vt = VT_BSTR;
V.bstrVal = SysAllocString(L"Значение");
// операции с переменной V
VariantClear(&V);
```

Для выполнения математических и логических операций с переменными типа **VARIANT** используют функции, приведенные в приложении А, раздел «Математические функции». Для выполнения операций с переменной типа **VARIANT**, содержащей тип **Currency**, используют функции, приведенные в приложении А, раздел «Функции для типа Currency».

BSTR

BSTR (*Basic String*) — строка символов в кодировке *Unicode*. Длина строки в байтах (без учета завершающего нулевого символа) записывается перед строкой в виде 32-х разрядного двоичного числа. Строка **BSTR** может содержать любое количество нулевых символов, а в конце строки записывается дополнительное нулевое значение из двух байт, что позволяет использовать функции для работы с «широкими» строками в стиле Си. Переменная типа **BSTR** указывает на начало строки. На рисунке 2 в строке типа **BSTR** записано слово *Hello* длиной 5 символов.

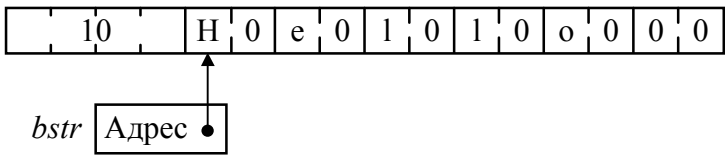

Рисунок 2 — Строка BSTR

Определение типа **BSTR** приведено в файле *wtypes.h* и имеет вид:

```
typedef OLECHAR FAR * BSTR;
```

Совместимым типом является **wchar_t ***.

Для работы со строками типа **BSTR** следует использовать системные функции, начинающиеся с **Sys**. Описание этих функций приведено в приложении Б.

Следующий пример показывает, как инициализировать **BSTR**, изменить значение, определить длину и освободить:

```
BSTR bstr;
bstr = SysAllocString(L"Hello");
SysReAllocString(&bstr, L"Good-bye");
UINT len = SysStringLen(bstr);
SysFreeString(bstr);
```

SAFEARRAY

В автоматизации используются массивы только типа **SAFEARRAY**, заданные в структуре **VARIANT**. **SAFEARRAY** — это структура, предназначенная для создания так называемых «безопасных» массивов (*safearray* — безопасный массив). Определение структуры приведено в файле *oaidl.idl*:

```
typedef struct tagSAFEARRAY {
    USHORT cDims;
    USHORT fFeatures;
    ULONG cbElements;
    ULONG cLocks;
    PVOID pvData;
    SAFEARRAYBOUND rgsabound[1];
} SAFEARRAY;
```

Здесь:

cDims — количество размерностей;

fFeatures — флаги, используемые функциями для **SAFEARRAY**; определены следующие биты:

FADF_AUTO (0x1) — массив размещается в стеке;

FADF_STATIC (0x2) — статический массив;

FADF_EMBEDDED (0x4) — массив внедрен в структуру;

FADF_FIXEDSIZE (0x10) — размерность массива постоянная;

FADF_RECORD (0x20) — массив содержит записи; если этот бит установлен, на смещении –4 относительно указателя на структуру находится указатель на интерфейс **IRecordInfo**;

FADF_HAVEIID (0x40) — массив имеет *IID*, идентифицирующий интерфейс; если этот бит установлен, на смещении –16 относительно указателя на структуру находится *GUID*; должен быть также установлен один из битов **FADF_UNKNOWN** или **FADF_DISPATCH**.

FADF_HAVEVARTYPE (0x80) — массив имеет заданный **VARIANT**-совместимый тип; если этот бит установлен, на смещении –4 относительно указателя на структуру находится тип элементов массива (**VT_XXX**);

FADF_BSTR (0x100) — массив строк **BSTR**;

FADF_UNKNOWN (0x200) — массив указателей на **IUnknown**;

FADF_DISPATCH (0x400) — массив указателей на **IDispatch**;

FADF_VARIANT (0x800) — массив типов **VARIANT**;

cbElements — размер элементов массива;

cLocks — количество текущих блокировок массива;

pvData — указатель на блок данных массива;

rgsabound — границы размерностей массива.

Информация об индексах размерностей массива описывается структурой **SAFEARRAYBOUND**, определение которой приведено в файле *oaidl.idl*:

```
typedef struct tagSAFEARRAYBOUND {
    ULONG cElements;
    LONG lLbound;
} SAFEARRAYBOUND;
```

Здесь **cElements** — количество элементов в размерности, **lLbound** — индекс нижней границы размерности. Индекс верхней размерности можно вычислить по формуле

```
lUbound = cElements - lLbound - 1;
```

Для выполнения операций с безопасными массивами следует использовать системные функции, начинающиеся с **SafeArray**.

Создание объекта автоматизации

Для создания объекта автоматизации в скриптовых языках, а также в языке *MSVB* и в некоторых других используются две специальные функции: **CreateObject** и **GetObject**.

Функция **CreateObject** принимает строковый параметр, указывающий на название сервера и название кокласса объекта автоматизации. Например, чтобы получить объект автоматизации приложения *MS Excel*, следует вызвать функцию **CreateObject** следующим образом (Пример приведен для языка *MSVB*):

```
Dim Q As Object
Set Q = CreateObject("Excel.Application")
```

Второй, необязательный параметр функции **CreateObject** указывает наименование сервера, на котором следует создать объект.

Наиболее распространенной ошибкой, которая возникает при обращении к функции **CreateObject** — 429 *«ActiveX component can't create object*» (невозможно создать объект автоматизации). Программист должен быть готов к возникновению этой ошибки, поскольку работа автоматизации основана на строковых значениях и позднем связывании. Результат вызова функции **CreateObject** всегда следует проверять. На языке *MSVB* проверить наличие ошибки можно проверить, например, так:

```
Dim Q As Object
On Error Resume Next
Set Q = CreateObject("Excel.Application")
If (Err.Number <> 0) Then
    ' обработка ошибки
End If
```

Функция **CreateObject** создает объект автоматизации, загружая в память сервер объекта. Если сервер объекта уже загружен, создается еще одна копия сервера. Например, пользователь работает с *MS Excel*, а в то же время контроллер автоматизации требует объект автоматизации этого приложения. Чтобы получить объект, не загружая сервер, используют функцию **GetObject**. Например, на языке *MSVB* получить объект от загруженного сервера можно при помощи следующего кода:

```
Dim Q As Object
On Error Resume Next
Set Q = GetObject(, "Excel.Application")
If (Err.Number <> 0) Then
    Set Q = CreateObject("Excel.Application")
End If
```

Функцию **GetObject** используется также для того, чтобы получить объект автоматизации, используя файл данных (документ) сервера автоматизации. В следующем примере создается объект автоматизации типа **Excel.Workbook** (а не типа **Excel.Application**):

```
Set Q = GetObject("D:\Document\Phones.xls")
```

В заключении следует отметить, что при вызове функции **GetObject** следует указать либо первый, либо второй, либо оба параметра.

Связывание

Связывание — это (упрощенно) процесс вычисления адреса функции. В классическом объектно-ориентированном программировании различают *раннее* и *позднее* связывание.

Ранее связывание означает вычисление адреса на этапе компиляции программы, при котором адрес функции вычисляется компилятором и записывается непосредственно в машинный код (в инструкцию **`CALL`**).

При *позднем связывании*, которое возникает при использовании виртуальных функций, адрес функции вычисляется во время работы программы при обращении к функции.

С точки зрения классического программирования связывание в автоматизации всегда *позднее*, поскольку автоматизация, как и все технологии на основе *COM*, функционируют на основе интерфейсов, представляющих собой наборы чистых виртуальных функций. Тем не менее, в автоматизации различают свои раннее, среднее и позднее связывание.

Раннее связывание в автоматизации (*early binding*) возникает, когда используется пользовательский интерфейс объекта автоматизации, и соответствует связыванию в любой другой *COM*-технологии. Например, проект в среде *MSVB* может иметь ссылку на сервер объекта автоматизации. При этом в среде работает интерактивная подсказка, позволяющий написать код без ошибок, но результирующая программа будет работать только если в системе установлен сервер объекта автоматизации. Для выполнения раннего связывания автоматизации класс автоматизации должен экспортировать пользовательский интерфейс.

Позднее связывание в автоматизации (*late binding*) возникает при использовании функций **`CreateObject`** или **`GetObject`** для создания объекта автоматизации. Собственно связывание выполняет метод **`Invoke`**, вызову которого предшествует вызов метода **`GetIDsOfNames`**. Позднее связывание автоматизации значительно замедляет вызов методов и обращение к свойствам объекта автоматизации, — намного больше, чем классическое позднее связывание.

Средним связыванием (или связыванием через *dispid*) в автоматизации называют позднее связывание, в котором обращение к методу **`GetIDsOfNames`** или другим методам интерфейса **`IDispatch`** (за исключением метода **`Invoke`**) отсутствует. В этом случае вызов метода пользовательского интерфейса немного ускоряется. Предпосылкой для среднего связывания является наличие информации о типах и идентификаторах *dispid*, которую разработчик контроллера может иметь на момент разработки.

Дуальный интерфейс

На рисунке 4 показана таблица виртуальных методов *vtable* для обычного *COM*-объекта, пользовательский (*custom*) интерфейс которого наследуется от **`IUnknown`**.

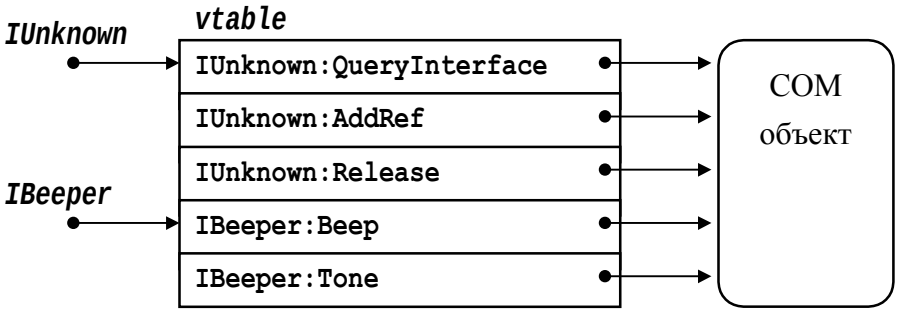

Рисунок 4 — *vtable* для *custom*-интерфейса

В качестве примера здесь используется объект с пользовательским интерфейсом **`IBeeper`**, имеющий метод **`Beep`** и свойство **`Tone`**.

Для автоматизации какого-либо приложения необходимо, чтобы оно экспортировало объектную модель, или как минимум класс, поддерживающий интерфейс диспетчеризации **`IDispatch`**. На рисунке 5 приведена таблица виртуальных методов *vtable* для интерфейса диспетчеризации.

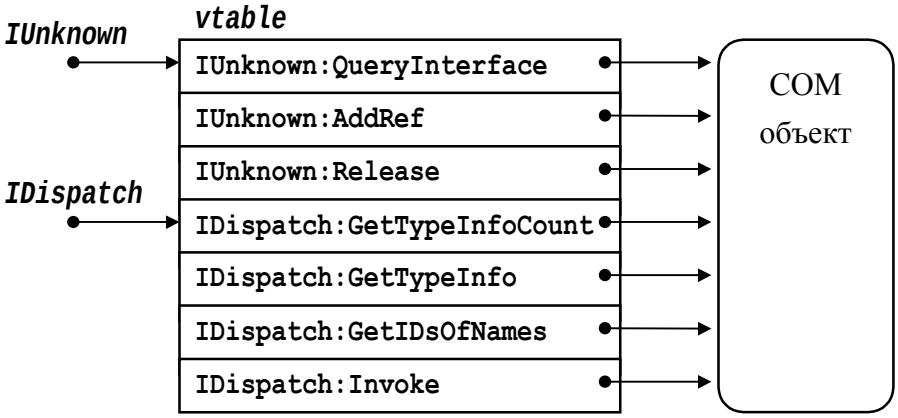

Рисунок 5 — *vtable* для чистого объекта автоматизации

Объект автоматизации, экспортирующий только интерфейс автоматизации, называется *чистым объектом автоматизации*. Для чистого объекта автоматизации можно выполнить только позднее или среднее связывание, что в значительной мере замедляет выполнение вызовов. Функциональность индивидуального объекта в этом случае скрыта, то есть реали-

зована с помощью закрытых методов и свойств класса автоматизации, «спрятанных» на рисунке 5 в прямоугольнике «*Объект*».

Интерфейс диспетчеризации используют в скриптовых языках или в проекте *MSVB*. При этом используется позднее связывание, которое значительно замедляет выполнение программ. Чтобы ускорить выполнение вызовов там, где это возможно, объект автоматизации должен экспортировать также свой пользовательский интерфейс. В «продвинутых» языках, таких, как *MSVC++*, а также в случае, когда есть возможность установить ссылку на сервер автоматизации в проекте *MSVB*, для ускорения выполнения вызовов лучше использовать раннее связывание. Для того, чтобы раннее связывание было возможно, необходимо, чтобы класс автоматизации поддерживал не только интерфейс диспетчеризации, но имел также собственный, пользовательский интерфейс.

На практике чаще всего используется не дополнительный пользовательский интерфейс, наследуемый от **`IUnknown`**, а так называемый *дуальный* (*dual*) интерфейс, наследуемый от **`IDispatch`** (рисунок 6).

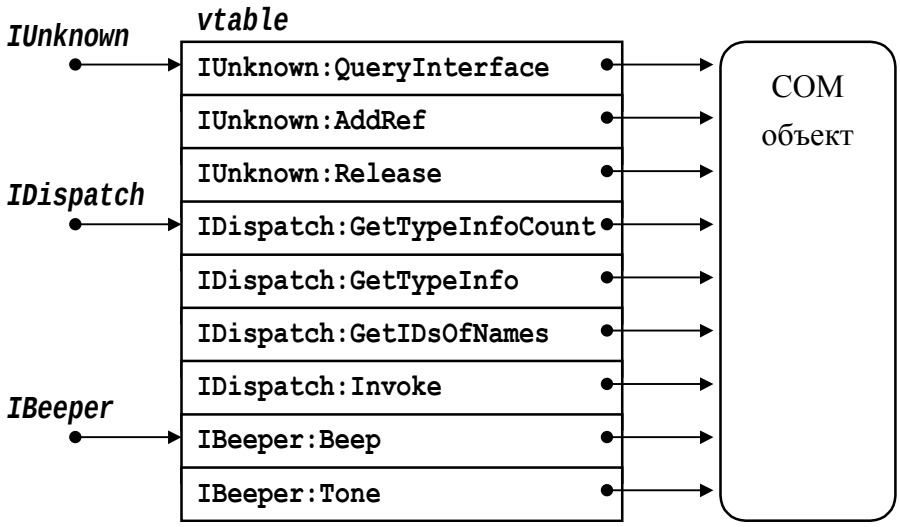

Рисунок 6 — *vtable* для дуального интерфейса

Таблица виртуальных методов *vtable* дуального интерфейса содержит вхождения сразу для трех интерфейсов в следующем порядке: **`IUnknown`**, **`IDispatch`**, *custom*-интерфейс. Это дает возможность получить как позднее связывание через интерфейс диспетчеризации, так и ранее связывание через *vtable*. Дуальность (двойственность) интерфейса проявляется в возможности применения как раннего, так и позднего связывания.

Скрипты

Скриптом называется программный код, не требующий предварительной компиляции. Для исполнения скрипта на компьютере должна быть установлена исполняющая система (хост), которая выполняет его интерпретацию и исполнение. В данном разделе рассматриваются скрипты *Windows*, написанные на языке *Visual Basic*. Скрипты Windows предназначены для автоматизации работ по управлению компьютером при помощи технологии *OLE Automation*. Скрипты на языке *Visual Basic Scripting Edition* имеют расширение **.vbs** (*Visual Basic Script*).

Скрипты на языке *Visual Basic Scripting Edition* имеют несколько ограничений по отношению к языку программирования *Visual Basic*. Прежде всего, при объявлении переменных нельзя указывать тип (подразумевается тип **Variant**). Использовать ключевые слова **Private** и **Public** можно, но не имеет смысла, так как все объекты языка имеют область действия в пределах только одного модуля — самого скрипта. Константы и типы, доступные в проекте *Visual Bas*ic, в скрипте отсутствуют. Каждая необходимая константа должна быть описана явным образом.

Код скрипта исполняется по ходу текста, по мере обнаружения операторов языка, расположенных вне процедур. Текст скрипта может содержать процедуры в произвольном порядке и месте расположения. Эти процедуры могут быть вызваны как из самих процедур, так и из любого оператора, расположенного вне процедур.

Система *Windows* имеет две исполняющие системы для выполнения скриптов *Windows* — **cscript** и **wscript**. Хост **cscript** исполняет скрипт под управлением командной строки, а хост **wscript** исполняет скрипт в графической оболочке. Хост указывается первым параметром командной строки, если она используется для запуска скрипта. По умолчанию, а также при запуске скрипта из оболочки *Explorer* (*Проводник*) используется хост **wcsript**. В следующем примере из командной строки запускается скрипт с именем **a.vbs** с четырьмя параметрами в исполняющей системе **cscript**:

```
cscript a.vbs first second 25 hello
```

Скрипты *Windows* могут иметь параметры (аргументы) командной строки. Каждый отдельный параметр записывается как элемент коллекции **Arguments** исполняющей системы **WScript**. В следующем примере на терминал выводятся все параметры командной строки перебором элементов коллекции при помощи цикла **For**…**Each**:

```
Dim A, I
WScript.Echo "Параметры скрипта:" & vbNewLine
For Each A In WScript.Arguments
    I = I + 1
    WScript.Echo CStr(I) & ": " & A
Next
```

Для вывода сообщений на терминал используется объект исполняющей системы **WScript** и метод **Echo**. В зависимости от того, какой исполняющей системой выполняется скрипт, сообщения выводятся на терминал (**cscript**) или в диалоговое окно (**wscript**).

Для создания объекта автоматизации в скрипте так же используются функции **CreateObject** или **GetObject**. В следующем примере в скрипте создается объект *MS Excel*:

```
Dim Q
Set Q = CreateObject("Excel.Application")
Q.Visible = True
```

По завершении работы скрипта объектная **Q** переменная автоматически освобождается, однако объект *MS Excel* продолжает работать.

В следующем примере показано, как в скрипте используются процедуры:

```
Dim Q
Set Q = CreateObject("Excel.Application")
Q.Visible = True
DoSomeActions
Q.Quit
Sub DoSomeActions
    Dim A
    Set A = Q.Workbooks.Add
    A.Worksheets(1).Cells(1, 1).Value = 255
End Sub
```

Здесь процедура **DoSomeActions** может быть описана как в начальной части скрипта, так и в конечной. Вызов процедуры не обязательно должен предшествовать ее описанию.

Объектная файловая модель

Объектная файловая модель *Microsoft Scripting Runtime* (*scrrun.dll*) предназначена для взаимодействия с файловой системой из скриптов. Она состоит из классов и коллекций, представляющих диски, каталоги, подкаталоги, файлы, текстовые потоки и т.п.

Работа с объектной файловой моделью начинается с создания объекта **FileSystemObject** (в примерах для ясности приводятся типы переменных, хотя в скриптах это недопустимо):

```
Dim fso As FileSystemObject
Set fso = CreateObject("Scripting.FileSystemObject")
```

При помощи объекта **FileSystemObject** непосредственно можно выполнять операции с файлами и папками при помощи следующих методов:

CopyFile(Источник, **Приемник)** — копирует файл источник в приемник;
CopyFolder(Источник, **Приемник)** — копирует каталог источник в приемник;
CreateFolder(Спецификация) — создает каталог;
DeleteFile(Спецификация) — удаляет файл;
DeleteFolder(Спецификация) — удаляет каталог;
MoveFile(Откуда, **Куда)** — перемещает файл;
MoveFolder(Откуда, **Куда)** — перемещает каталог.

При помощи метода **FileExists** можно убедиться в существовании указанного файла, а при помощи метода **FolderExists** можно убедиться в существовании указанного каталога, например:

```
If FileExists("C:\autoexec.bat") Then КАКИЕ-ТО ДЕЙСТВИЯ
```

Диски

Единственное свойство объекта **FileSystemObject** — свойство **Drives** — возвращает коллекцию дисков. В следующем примере показан перебор всех дисков при помощи цикла **For**…**Each** и определение их имени, типа и метки тома при помощи соответствующих свойств:

```
Dim D As Drive, DC As Drives, S As String
Set DC = fso.Drives
For Each D in DC
    Debug.Print "Имя диска: " & D.DriveLetter
    Debug.Print "Тип устройства: " & D.DriveType
    Debug.Print "Метка тома: " & D.VolumeName
Next
```

Тип диска определяется одной из следующих констант:

UnknownType = **0** — неизвестно (не определено);
Removable = **1** — сменный (например, дискета или флэш-диск);
Fixed = **2** — фиксированный (жесткий);
Remote = **3** — удаленный (сетевой);
CDRom = **4** — оптический (CD-ROM, DVD-ROM);
RamDisk = **5** — диск в памяти (RAM-диск).

Важными свойствами объекта **Drive** являются также:

AvailableSpace As Variant — размер доступного пространства;
FileSystem As String — возвращает тип файловой системы;
FreeSpace As Variant — возвращает размер свободного пространства;
TotalSize As Variant — возвращает полный размер диска.

Каталоги

Для работы с каталогами используется объект **Folder**, получить который можно при помощи метода **GetFolder** объекта **FileSystemObject**, например:

```
Dim Folder As Folder
Set Folder = fso.GetFolder("C:\Windows")
```

Все подкаталоги данного каталога возвращает объект **Folders**, получить который можно при помощи свойства **SubFolders**. Далее отдельные подкаталоги можно получить при помощи цикла **For Each**, например:

```
Dim TheFolder As Folder
For Each TheFolder In Folder.SubFolders
    Debug.Print TheFolder.Name
Next
```

Важными свойствами объекта **Folder** являются:

Attributes — атрибуты каталога (см. ниже значения атрибутов);
Drive — объект **Drive**, описывающий содержащий папку диск;
Files — возвращает коллекцию файлов каталога;
IsRootFolder — признак, указывающий на корневой каталог;
Name — название каталога;
ParentFolder — возвращает объект **Folder** родительского каталога;
Path — спецификация каталога (свойство по умолчанию);
Size — размер каталога вместе с подкаталогами.

Атрибуты каталога (а равно и файла) являются суммой следующих констант (не все атрибуты применимы ко всем файловым системам):

Normal = **0**
ReadOnly = **1**
Hidden = **2**
System = **4**
Volume = **8**
Directory = **16**
Archive = **32**
Compressed = **2048**

Alias = **1024**

Методы объекта **Folder** выполняют действия с каталогом — **Copy** (скопировать), **Delete** (удалить), **Move** (переместить).

Файлы

Получить объект **File**, представляющий файл, можно при помощи метода **GetFile** объекта **FileSystemObject**, например:

```
Dim File As File
Set File = fso.GetFile("c:\autoexec.bat")
```

Все файлы каталога можно получить при помощи свойства **Files** объекта **Folder**, которое возвращает коллекцию всех файлов, например:

```
Dim TheFile As File
For Each TheFile In fso.GetFolder("C:\").Files
    Debug.Print TheFile.Name
Next
```

Объект **File** обладает следующими важными свойствами:

Attributes — атрибуты файла (описаны выше);
Drive — объект **Drive**, описывающий содержащий файл диск;
Name — название файла;
ParentFolder — возвращает объект **Folder** родительского каталога;
Path — спецификация файла (свойство по умолчанию);
Size — размер файла.

Объект **File** обладает теми же методами, что указаны выше для объекта **Folder**, а также методом **OpenAsTextStream**, который возвращает объект **TextStream**.

Потоки

Объект **TextStream** предназначен для выполнения операций с текстовыми файлами. Его можно получить, кроме указанного выше способа, при помощи метода **CreateTextFile** объектов **FileSystemObject** и **Folder**, а также при помощи метода **OpenTextFile** объекта **FileSystemObject**.

Метод **CreateTextFile** создает текстовый поток, при этом на диске создается указанный первым параметром текстовый файл. Если файл существует, то по умолчанию он будет перезаписан, например:

```
Dim ts As TextStream
Set ts = fso.CreateTextFile("C:\autoexec.bat")
```

Метод **OpenTextStream** создает текстовый поток из существующего или нового (по умолчанию) файла аналогичным образом. По умолчанию создается поток для чтения. При необходимости режим открытия указывается вторым параметром, который может принимать одно из следующих значений:

ForReading = **1** — файл открывается для чтения;
ForWriting = **2** — файл открывается для записи;
ForAppending = **8** — файл открывается для добавления;

Важные методы объекта **TextStream**:

Close — закрыть поток и файл на диске;
Read — прочитать указанное количество символов;
ReadAll — прочитать весь файл;
ReadLine — прочитать строку;
Write — записать в поток строку;
WriteLine — записать в поток строку и символ конца строки.

Свойства объекта **TextStream**:

AtEndOfLine — признак положения текущей точки в конце строки;
AtEndOfStream — признак положения текущей точки в конце потока;
Column — текущий номер позиции в строке;
Line — текущая строка в потоке.

В следующем примере показано, как открыть файл в виде потока и построчно вывести его в окно отладки *Immediate*:

```
Dim ts As TextStream
Set ts = fso.OpenTextFile("C:\boot.ini")
Do While Not ts.AtEndOfStream
    Debug.Print ts.ReadLine
Loop
```

В следующем примере тот же файл читается целиком в строковую переменную, которая и выводится в окно *Immediate*:

```
Dim ts As TextStream, S As String
Set ts = fso.OpenTextFile("C:\boot.ini")
S = ts.ReadAll
Debug.Print S
```

Автоматизация Microsoft Office

Microsoft Excel

Microsoft Excel — одно из лучших автоматизированных приложений в современном программном обеспечении. Сокращенная объектная модель *Microsoft Excel* приведена на рисунке 7 (основные классы):

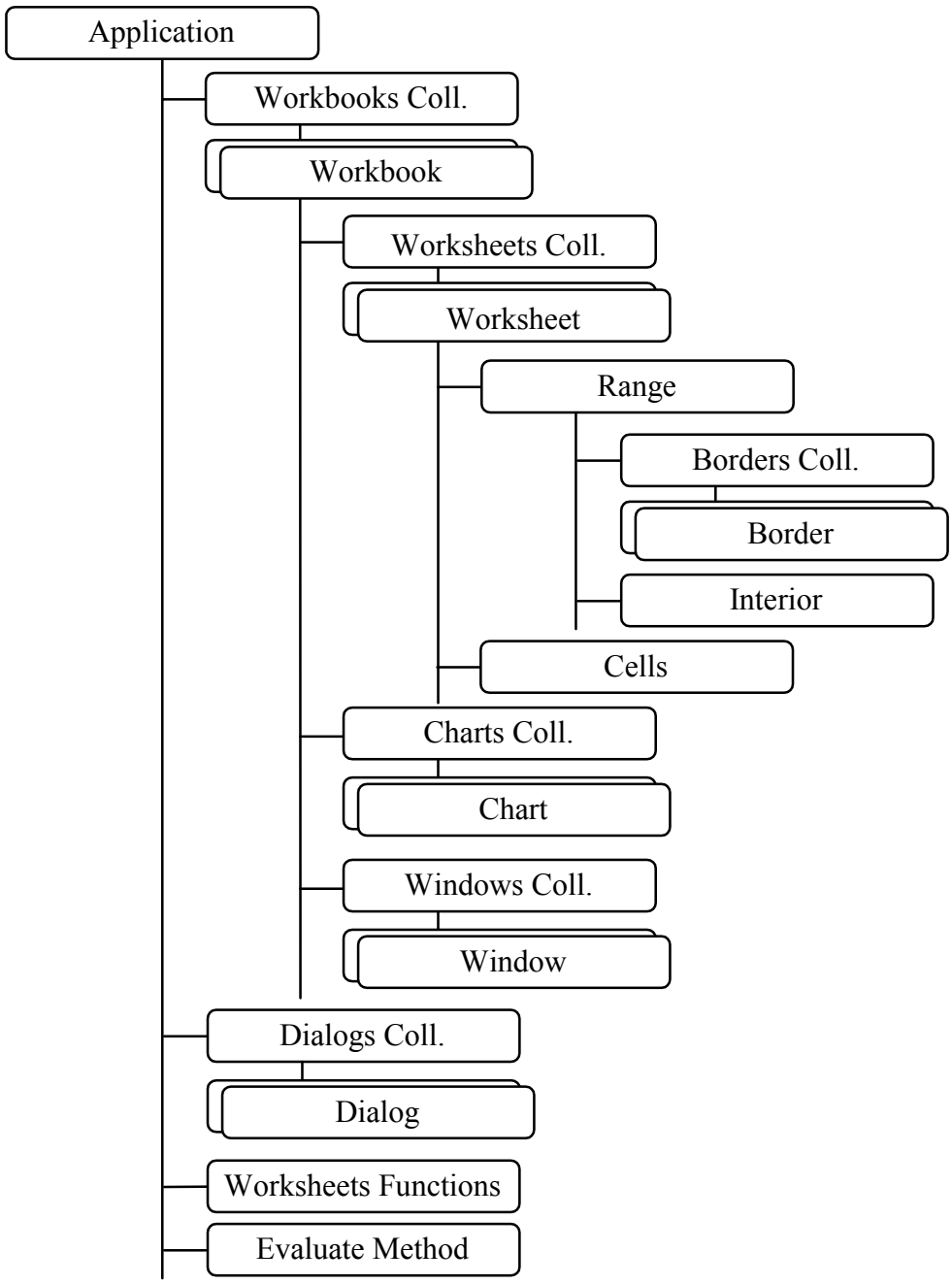

Рисунок 7 — Краткая объектная модель Microsoft Excel

Подробную объектную модель см. справку *Microsoft Excel*.

Express-справка

Приложение *MS Excel* состоит из рабочих книг (объектов **Workbook**), составляющих коллекцию **Workbooks**.

Рабочая книга, в свою очередь, состоит из листов. Листы могут иметь разный тип. Основной тип листа — расчетный (объект **Worksheet**). Все расчетные листы составляют коллекцию **Worksheets**. Другой распространенный тип листа — диаграмма (объект **Chart**). Все диаграммы составляют коллекцию **Charts**.

Для работы с отдельными ячейками расчетного листа используются объект **Range** и свойство **Cells**. Объект **Range** представляет собой диапазон ячеек, а свойство **Cells** возвращает объект **Range** для одной ячейки.

Для автоматизации *MS Excel* прежде всего создается объект автоматизации при помощи функции **CreateObject**:

```
On Error Goto CreateError
Dim EA As Object
Set EA = CreateObject("Excel.Application")
```

Сразу после активизации приложение невидимо и работает в фоновом режиме. Для того чтобы увидеть приложение (показать его), используется свойство **Visible**:

```
EA.Visible = True
```

Созданный объект представляет приложение, в котором нет рабочих книг, поэтому следующим действием является добавление новой рабочей книги и получение ссылки на нее:

```
Dim EB As Object
Set EB = EA.Workbooks.Add
```

Рабочая книга содержит минимум один расчетный лист, поэтому далее можно сразу получить ссылку на первый расчетный лист:

```
Dim ES As Object
Set ES = EB.Worksheets(1)
```

Далее выполняются операции с ячейками расчетного листа при помощи объекта **Range**. Дополнительно о создании объекта **Excel.Application** см. раздел «Создание объекта автоматизации».

Закрывается приложение при помощи метода **Application::Quit**.

Управление приложением

Управление приложением включает в себя действия, относящиеся к приложению в целом, через свойства и методы объекта **Application**.

Сразу после создания объекта автоматизации приложение невидимо. Для управления видимостью приложения используется свойство **Visible**. Показывать приложение во многих случаях нет необходимости, однако во время разработки контроллера это полезно для визуального удостоверения в правильности выполняемых действий.

Чтобы обеспечить полностью автономную работу контроллера (исключающую участие пользователя), полезно также отключить диалоги, которые приложение *MS Excel* показывает при выполнении некоторых операций, требуя ответа на те или иные вопросы. Сделать это можно при помощи свойства **DisplayAlerts**:

```
EA.DisplayAlerts = False
```

Однако разработчик контроллера должен понимать потенциальную опасность отключения диалогов *MS Excel*. Например, можно случайно заменить файл существующей книги другим при использовании метода **SaveAs** объекта **Workbook**.

Управление приложением включает в себя также управление диалогами. Все диалоги составляют коллекцию **Dialogs**, получить которую можно через одноименное свойство. В следующем примере используется диалог открытия файла книги:

```
Const xlDialogOpen = 1
EA.Dialogs(xlDialogOpen).Show
```

При этом потребуется участие пользователя для поиска и выбора файла книги. Перечень констант, указывающих на диалоги, можно найти при помощи *Object Browser* среды разработки *MSVB*.

Для выполнения вычислений при помощи *MS Excel* можно использовать не только расчетные листы, но и метод **Evaluate**:

```
Dim V As Variant
V = EA.Evaluate(выражение)
```

Здесь вместо «выражение» следует подставить выражение. Это может быть либо строковый литерал, либо строковая переменная. Обратим внимание, что результат вычислений всегда следует принимать в переменную типа **Variant**, так как он может содержать нечисловые значения, такие, на-

пример, как сообщение об ошибке. Поэтому полученный результат следует проверять, например, так:

```
V = EA.Evaluate(выражение)
If IsError(V) Then
    ' Обработка ошибки
Else
    ' Обработка результата
End If
```

Выражение может также содержать вызовы встроенных функций *MS Excel*. В следующем примере выражение содержит обращение к функциям **Sin** и **Radians**:

```
Dim V As Variant, Result As String
V = EA.Evaluate(SIN(RADIANS(30)))
Result = CStr(V)
```

Может оказаться, что в одних случаях названия функций следует использовать английские, а в других — русские. В случае, если при вычислении выражения, которое содержит вызов функции, возвращается ошибка 2029, скорее всего название функции записано не на том языке.

Управление книгами

Управление книгами включает в себя открытие и сохранение книг, добавление новой книги, выбор книги из коллекции. Большая часть операций выполняется коллекцией **Workbooks**, а сохранение книги выполняется самой книгой — объектом **Workbook**.

Новую книгу добавляет метод **Add** коллекции **Workbooks**:

```
Set EB = EA.Workbooks.Add
```

Метод **Add** имеет один необязательный параметр — шаблон книги. Это либо спецификация файла шаблона, либо константа, указывающая на тип листа в новой книге. Лист в книге в этом случае будет единственным. Допустимые константы:

xlWBATWorksheet = -4167 (&HFFFFEFB9) — расчетный лист;

xlWBATChart = -4109 (&HFFFFEFF3) — лист диаграммы;

xlWBATExcel4IntlMacroSheet = 4 — лист макроса;

xlWBATExcel4MacroSheet = 3 — лист макроса.

В примере создается новая книга с одним расчетным листом:

```
Set EB = EA.Workbooks.Add(-4167)
```

Если ссылка на книгу **EB** не нужна, то можно использовать оператор **With** для операций с ней, например, так:

```
With EA.Workbooks.Add
    Set ES = .Worksheets(1)
End With
```

Чтобы открыть книгу, сначала нужно каким-то образом получить спецификацию файла книги **Path**, а затем использовать метод **Open**:

```
Set EB = EA.Workbooks.Open(Path)
```

Метод **Open** имеет множество необязательных параметров, определяющих режим открытия. Например, можно открыть книгу, создав ее из текстового файла, в котором каждая строка содержит несколько полей, разделенных каким-либо специальным знаком. В примере в качестве знака разделителя используется знак номера **#**.

```
Path = "D:\Document\1.txt"
Set EB = EA.Workbooks.Open(Path, , , 6, , , , , "#")
```

Здесь четвертый параметр **6** обозначает, что открывается текстовый файл, в котором в качестве разделителя полей используется знак, указанный девятым параметром **#**.

Текстовый файл можно также открыть методом **OpenText**:

```
EA.Workbooks.OpenText Path, , , , , , , , , , True, "#"
```

При этом объектная ссылка на книгу не создается. Ссылка не создается также в случае, когда для открытия книги используется диалог открытия самого приложения *MS Excel*:

```
Const xlDialogOpen = 1
EA.Dialogs(xlDialogOpen).Show
```

При этом также требуется участие пользователя для управления диалогом «Открыть» для поиска и выбора открываемого файла. Подробнее о параметрах метода **Open** см. справку *MS Excel*.

Для сохранения книги, которая до этого не сохранялась ни разу, используется метод **SaveAs** объекта **Workbook**:

```
EB.SaveAs Path, xlWorkbookNormal
```

При этом каким-либо образом нужно получить спецификацию файла для сохранения. Если книга уже сохранялась, то при использовании метода **SaveAs** *MS Excel* может показать диалог запроса о перезаписи, на который

должен ответить пользователь. Как исключить этот диалог, описано в разделе «Управление приложением». В примере в качестве параметра указана константа, предписывающая сохранить документ как книгу *MS Excel*. Подробнее о параметрах метода см. справку *MS Excel*.

Если книга хотя бы раз сохранялась, повторно сохранить ее можно при помощи метода **Save**. Метод не имеет параметров.

Выбор той или иной книги и получение ссылки на нее производится при помощи свойства **Item** коллекции **Workbooks**. Поскольку свойство **Item** является свойством по умолчанию в любой коллекции, название свойства можно не указывать. Проще всего получить ссылку на книгу, указав ее порядковый номер в коллекции:

```
Dim EB As Excel.Workbook
Set EB = EA.Workbooks(1)
```

Выбрать книгу можно также, указав ее наименование (имя файла):

```
Dim EB As Excel.Workbook
Set EB = EA.Workbooks("phones")
```

Управление листами

Управление листами заключается в операциях их добавления, удаления, перемещения, копирования, переименования, а также получении ссылки на тот или иной лист.

Добавление нового листа в книгу осуществляется при помощи метода **Add** коллекции **Worksheets**:

```
Set ES = EB.Worksheets.Add
```

Параметры метода **Add** указывают, куда, сколько и каких листов добавить. Первые два параметра указывают положение нового листа (листов). Первый параметр указывает номер или имя листа, перед которым следует добавить новый лист (листы). Второй параметр указывает номер или имя листа, после которого следует добавить новый лист (листы). Эти два параметра являются взаимно исключающими. Третий параметр указывает количество добавляемых листов, а четвертый — их тип.

Удаление листа осуществляется при помощи метода **Delete** объекта **Worksheet**. В примере удаляется последний лист:

```
EB.Worksheets(EB.Worksheets.Count).Delete
```

Для перемещения листа используется метод **Move** объекта **Worksheet**. В примере первый лист перемещается в конец книги:

```
EB.Worksheets(1).Move , EB.Worksheets(EB.Worksheets.Count)
```

Для копирования листа используется метод **Copy**. В примере первый лист копируется в конец книги:

```
EB.Worksheets(1).Copy , EB.Worksheets(EB.Worksheets.Count)
```

Получить ссылку на лист книги можно при помощи свойства **Item** коллекции **Worksheets**. В качестве параметра указывается либо порядковый номер листа в коллекции, либо его наименование. В примере используется порядковый номер листа:

```
Set ES = EB.Worksheets(1)
```

В следующем примере лист выбирается по своему наименованию, указанному на ярлычке:

```
Set ES = EB.Worksheets("Лист1")
```

Работа с ячейками

Для доступа к содержимому ячеек используется объект **Range** и его свойства **Formula** и **Value**. Следует помнить о том, что свойство **Cells** также возвращает объект **Range**. Установка значений ячеек не представляет особой трудности. Вот несколько примеров:

```
ES.Range("A1").Value = 256
ES.Cells(2, 3).Value = "Hello"
ES.Range("A1:A5").Formula = "=Rand()"
```

При использовании свойства **Range** ячейка или диапазон ячеек задается строковым значением. При этом значение типа **"A1"** задает одну ячейку, а значение типа **"A1:C5"** — диапазон. При использовании свойства **Cells** задается одна ячейка числовыми значениями номера строки и столбца.

При чтении содержимого ячеек следует помнить о том, что в большинстве случаев возвращаемое значение должно приниматься в переменную типа **Variant**, так возвращаемое значение не обязательно является числовым или строковым, как предполагается:

```
Dim V As Variant
V = ES.Range("A1").Value
```

После чтения значения небесполезно проанализировать полученное значение на предмет того, какого типа значение содержит переменная **Variant**. Это осуществляется при помощи функций типа **IsXXXX** — **IsArray**, **IsDate**, **IsEmpty**, **IsError**, **IsNull**, **IsNumeric**, **IsObject**, например, так:

```
If IsNumeric(V) Then
ElseIf IsError(V) Then
End If
```

Особый интерес возникает при работе с диапазонами ячеек и массивами. При чтении диапазона переменная типа **Variant** получает значение массива:

```
Dim V As Variant
V = EB.Worksheets(1).Range("A1:A5")
Dim A As Double
A = V(1, 1)
A = V(2, 1)
A = V(3, 1)
```

Следует обратить внимание на то, что возвращаемый массив всегда двухмерный. В следующем примере в переменную принимается двухмерный массив ячеек:

```
Dim V As Variant
V = EB.Worksheets(1).Range("A1:B5")
Dim A As Double
A = V(1, 1)
A = V(1, 2)
A = V(2, 1)
A = V(2, 2)
```

Первый индекс массива соответствует строке, второй — столбцу.

Аналогично можно присвоить значение массива диапазону ячеек:

```
Dim V As Variant
ReDim V(1 To 5, 1 To 1) As Integer
V(1, 1) = 1
V(2, 1) = 2
V(3, 1) = 3
V(4, 1) = 4
V(5, 1) = 5
ES.Range("A1:A5") = V
```

Заметим, что индексы массива должны начинаться с единицы и массив должен быть двухмерным в любом случае.

Оформление ячеек

В оформление ячеек входят границы (рамки), цвет рамки, фона и шрифта, а также выравнивание текста. Управление границами осуществляется при помощи свойства **Borders** объекта **Range**, которое возвращает одноименную коллекцию из 8 объектов **Border** — границ. Для каждой границы в *MS Excel* определена константа:

xlDiagonalDown = **5** — диагональная вниз;

xlDiagonalUp = **6** — диагональная вверх;

xlEdgeLeft = **7** — левая;

xlEdgeTop = **8** — верхняя;

xlEdgeBottom = **9** — нижняя;

xlEdgeRight = **10** — правая;

xlInsideVertical = **11** — вертикальная внутри;

xlInsideHorizontal = **12** — горизонтальная внутри.

Тип линии рамки задается при помощи свойства **LineStyle** объекта **Border**. Для задания типа рамки используются константы:

xlContinuous = **1** — сплошная;

xlDash = **-4115** (**&HFFFFFEFD**) — штриховая;

xlDashDot = **4** — штрих пунктирная;

xlDashDotDot = **5** — штрих-два пунктира;

xlDot = **-4118** (**&HFFFFFEFA**) — пунктирная;

xlDouble = **-4119** (**&HFFFFFEF9**) — двойная;

xlLineStyleNone = **-4142** (**&HFFFFFEFD2**) — отсутствует;

xlSlantDashDot = **13** — наклонный штрих-пунктир.

Толщина рамки задается при помощи свойства **Weight** объекта **Border**. Для задания толщины используются константы:

xlHairline = **1** — очень тонкая;

xlThin = **2** — тонкая;

xlThick = **4** — толстая;

xlMedium = **-4138** (**&HFFFFFEFD6**) — средняя.

Цвет рамки задается свойствами **Color** и **ColorIndex** объекта **Border**. Для задания цвета при помощи свойства **Color** используется непосредственное числовое значение. Для задания цвета при помощи свойства **ColorIndex** используется число от 1 до 56, задающее один из 56 заранее определенных цветов, например: 1 — черный, 2 — белый.

Цвет шрифта в ячейке управляется свойствами **Color** и **ColorIndex** объекта **Range.Font**, а цвет фона ячейки — свойствами **Color** и **ColorIndex** объекта **Range.Interior**.

Положение текста в ячейке по горизонтали задается свойством **HorizontalAlignment**, а по вертикали — свойством **VerticalAlignment** объекта **Selection**.

Комплексный пример задания рамки и цвета рамки, шрифта и фона, а также выравнивания текста:

```
With ES.Range("A1")
    .Value = 123
    .Borders(xlEdgeBottom).LineStyle = xlDouble
    .Borders(xlEdgeBottom).Weight = xlMedium
    .Borders(xlEdgeBottom).ColorIndex = 3    ' Red
    .Interior.ColorIndex = 5                 ' Blue
    .Font.Color = &HF0F0F0                   ' Almost White
    .Select
    With EA.Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlCenter
    End With
End With
```

Microsoft Word

Microsoft Word является важнейшим приложением *Microsoft Office*, который представляет собой универсальный программный компонент для обработки текстов. Значительно сокращенная объектная модель (основные классы) этого приложения приведена на рисунке 8.

Объектная модель *MS Word* насыщена объектами, назначение которых не всегда сразу можно понять. Это вытекает из многообразия функций самого приложения, — откройте его и попробуйте подсчитать количество диалогов и параметров, устанавливаемых с их помощью. Стили, шаблоны, поля и свойства документа являются важными элементами, которые чаще всего остаются неиспользованными. Пользователь зачастую даже не догадывается о возможностях, которые *MS Word* может предоставить, и применяет лишь ограниченное подмножество функций. В связи со сложностью как объектной модели *MS Word*, так и с многообразием выполняемых приложением функций, описание полной объектной модели *MS Word* заслуживает отдельной книги. В данном пособии рассматриваются только начальные сведения по управлению приложением *MS Word*.

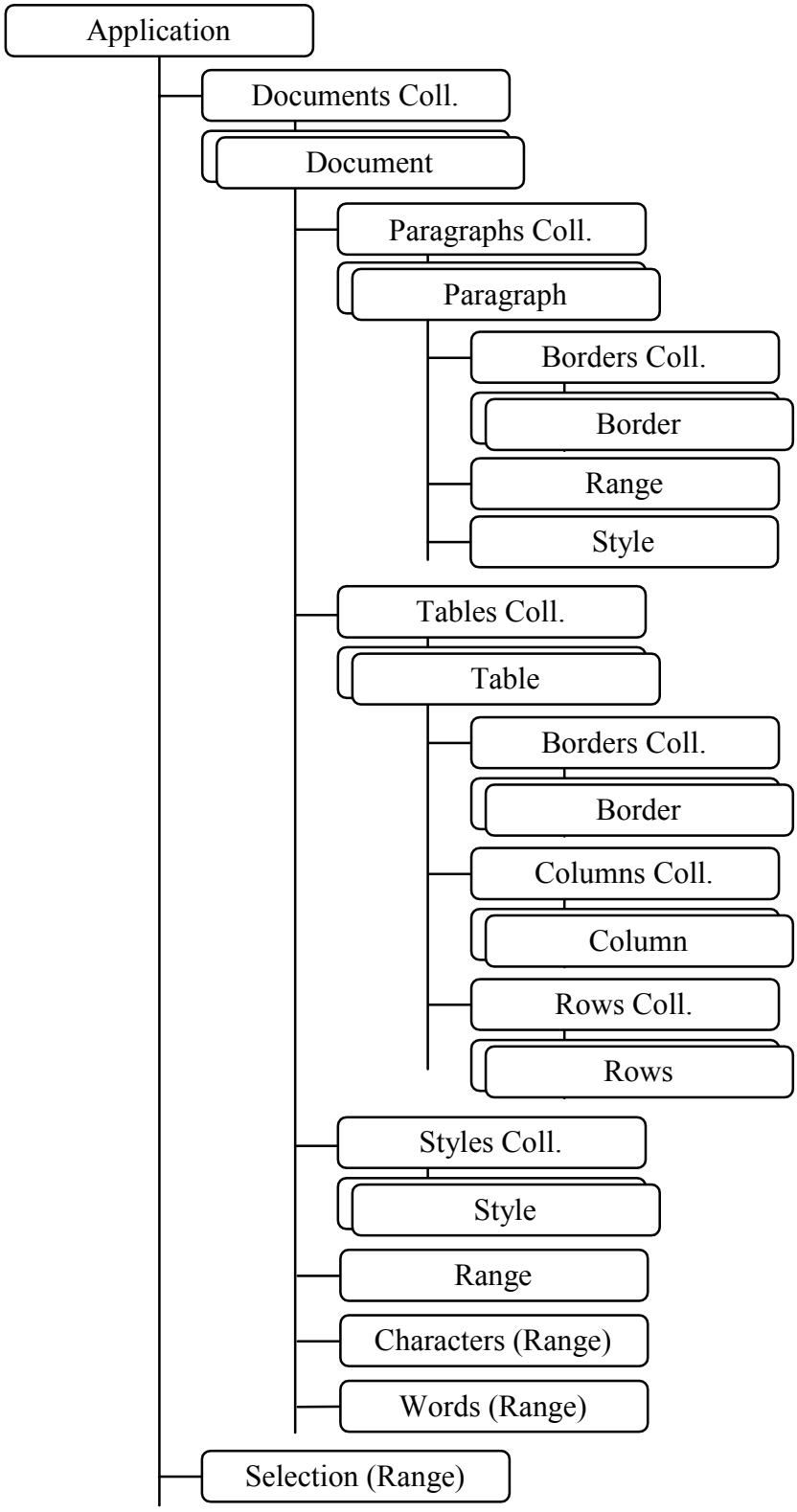

Рисунок 8 — Краткая объектная модель Microsoft Word

Основными элементами объектной модели *MS Word* являются объект **`Document`**, представляющий документ, объект **`Selection`**, представляющий выделенную часть документа, объект **`Range`**, представляющий часть доку-

мента, выбранную для выполнения действий. Объект **Selection**, и некоторые другие, например **Characters** и **Words**, возвращают объект **Range**.

Объект **Range** представляет собой важный элемент объектной модели. С его помощью выполняется большинство действий. С точки зрения пользовательского интерфейса объект **Range** не существует. Пользователь выполняет действия, выделяя часть документа. При этом формируется объект **Selection**, который возвращает объект **Range**. С точки зрения программного управления нет необходимости выполнять фактическое выделение части документа (формировать объект **Selection**). Вместо этого можно выполнить логическое выделение части документа, формируя объект **Range**.

Управление приложением

Управление приложением *MS Word* принципиально ничем не отличается от управления приложением *MS Excel*. Работа с приложением начинается с создания объекта автоматизации (в примере **WA**) при помощи функции **CreateObject** или **GetObject** обычным образом:

```
Dim WA As Object
Set WA = CreateObject("Word.Application")
```

Сразу после создания объект является невидимым. При помощи свойства **Visible** его можно отобразить на экране с целью контроля выполняемых действий, например, так:

```
WA.Visible = True
```

Отключить предупреждающие диалоги *MS Word* можно при помощи свойства **DisplayAlerts**, имеющее тип **Long**. Свойство может принимать одно из трех значений:

wdAlertsNone = 0 — отключить диалоги и сообщения об ошибках;

wdAlertsMessageBox = **-2** — отключить диалоги;

wdAlertsAll = -1 — включить диалоги и сообщения об ошибках.

Управление диалогами типа «Открыть» или «Сохранить как» производится при помощи коллекции **Dialogs**. В следующем примере используется диалог для выбора и открытия документа:

```
WA.Dialogs(wdDialogFileOpen).Show
```

Важными методами приложения являются функции пересчета единиц измерения, используемые для задания параметров страниц, параграфов и т.п. Есть следующие функции:

CentimetersToPoints — пересчитывает сантиметры в пункты;
InchesToPoints — пересчитывает дюймы в пункты;
LinesToPoints — пересчитывает строки в пункты (1 строка = 12 пунктов);
MillimetersToPoints — пересчитывает миллиметры в пункты;
PicasToPoints — пересчитывает пики в пункты (1 пика = 12 пунктов);

Аналогичным образом могут быть пересчитаны пункты в любую из приведенных единиц измерения:

PointsToCentimeters — пересчитывает пункты в сантиметры;
PointsToInches — пересчитывает пункты в дюймы;
PointsToLines — пересчитывает пункты в строки;
PointsToMillimeters — пересчитывает пункты в миллиметры;
PointsToPicas — пересчитывает пункты в пики;

Управление документами

Управление документами включает в себя создание нового документа, открытие существующего документа и сохранение.

Новый документ создается при помощи метода **Add** коллекции документов **Documents**:

```
Set WD = WA.Documents.Add
```

При этом создается новый документ на основе шаблона по умолчанию «Обычный» (**Normal.dot**). При необходимости создать документ на основе другого шаблона, он должен быть задан первым параметром метода **Add** (в виде спецификации файла шаблона). Второй параметр, если задан, указывает, как использовать шаблон. Если параметр равен значению **True**, то шаблон открывается как шаблон, иначе как документ на основе указанного шаблона.

Открыть документ можно при помощи указанного выше диалога, а также при помощи метода **Open** коллекции **Documents**, например:

```
Set WD = WA.Documents.Open(Path)
```

Здесь **Path** — это спецификация открываемого документа. Дополнительно к спецификации могут быть указаны подтверждение преобразования, открытие в режиме чтения, добавление документа в список недавно открывавшихся файлов, пароль для документа, пароль для шаблона, признак повторного открытия (если документ уже открыт), пароль для сохранения документа, пароль для сохранения шаблона и формат документа.

Формат документа задается одной из следующих констант:

wdOpenFormatAuto = **0** — автоматически определять формат;
wdOpenFormatDocument = **1** — использовать формат документа *Word*;
wdOpenFormatTemplate = **2** — открыть как шаблон;
wdOpenFormatRTF = **3** — использовать RTF формат;
wdOpenFormatText = **4** — использовать формат текстового документа;
wdOpenFormatUnicodeText = **5** — использовать формат текстового документа в кодировке *Unicode*.

Сохранить документ можно при помощи диалога «Сохранить как» и при помощи метода **Save** объекта **Document**. Во втором случае, если документ до этого ни разу не сохранялся, появится диалог «Сохранить как» для указания спецификации нового файла. Чтобы сохранить документ в другой файл, используется метод **SaveAs** объекта **Document**:

```
WD.SaveAs Path
```

Метод имеет десять дополнительных параметров сохранения.

Сохранить все открытые документы можно также при помощи метода **Save** коллекции **Documents**. При этом дополнительно указывается, предлагать пользователю сохранить документ, или автоматически сохранить изменения. Второй параметр этого метода указывает способ сохранения.

Закрыть приложение *MS Word* можно при помощи метода **Quit**.

Управление текстом

Основные действия в документе производятся над его текстом. Это может быть применение стиля, такого, как «Обычный» или «Заголовок 1», к абзацу, формата, такого, как начертание «полужирное» или «курсивное», к части текста, добавление нового текста, удаление текста, поиск и замена, проверка правописания, перемещение текущей точки (курсора) и многое другое.

В большинстве случаев для выполнения действий с текстом требуется выделить или выбрать его часть. Для этой цели используется объект **Range**, представляющий собой непрерывный блок документа. Объект **Range** имеет два свойства, которые указывают на начало (свойство **Start**) и конец (свойство **End**) блока.

В следующем примере показано, как выделить второй по порядку абзац текста (включая маркер конца абзаца):

```
WA.Selection.Start = WD.Paragraphs(2).Range.Start
```

```
WA.Selection.End = WD.Paragraphs(2).Range.End
```

Здесь при помощи метода **Range** и метода **Item(Index)** коллекции абзацев **Paragraphs** определяются точки начала и конца абзаца, которые задают начало и конец выделения текста в документе. Заметим, что свойство **Item** является свойством по умолчанию, а точки начала и конца выделения отсчитываются в символах, включая невидимые.

Далее над выделенным текстом можно выполнять разные действия, предусмотренные объектом **Range**. Приведем некоторые свойства объекта:

- свойства **Bold** и **Italic** управляют начертанием «полужирный» и «курсивный» соответственно;

- свойство **Case** управляет регистром букв. Это свойство может принимать следующие значения:

wdLowerCase = **0** — перевести в нижний регистр;

wdUpperCase = **1** — ПЕРЕВЕСТИ В ВЕРХНИЙ РЕГИСТР;

wdTitleWord = **2** — Каждое Слово С Прописной Буквы;

wdTitleSentence = **4** — Предложение с прописной буквы;

wdToggleCase = **5** — переключить регистр на противоположный.

- свойство **Characters** возвращает коллекцию одиночных символов;

- свойство **Font** возвращает объект **Font**, с помощью которого можно управлять всеми параметрами шрифта — гарнитурой, размером, начертанием, подчеркиванием и т.п.;

- свойство **LanguageID** управляет языком. Значение свойства равно идентификатору языка операционной системы. Например, русский язык имеет идентификатор 1049, английский — 1033;

- свойство **Style** управляет стилем. Значением этого свойства является название стиля, зарезервированная константа (для встроенных стилей) или объект **Style**;

- свойство **Text** возвращает или устанавливает чистый текст;

- свойство **Words** возвращает коллекцию слов.

Некоторые методы объекта **Range**:

- метод **InsertAfter** вставляет указанный текст в конец блока;

- метод **InsertBefore** вставляет указанный текст в начало блока;

- метод **InsertParagraph** заменяет блок новым параграфом;

- метод **Copy** копирует блок в буфер обмена;

- метод **Paste** вставляет блок из буфера обмена;

Операции с отдельными буквами или словами можно выполнять, используя коллекции **Characters** и **Words**. Элементы этих коллекций представляют собой также объекты **Range**.

Коллекция **Paragraphs** представляет собой все абзацы документа в виде объектов **Paragraph**. При помощи метода **Add** этой коллекции можно добавить новый абзац, например, так:

```
WD.Paragraphs.Add
```

При этом новый абзац добавляется в конец текста или выбранного блока. Чтобы добавить абзац в произвольное место, нужно указать объект **Range** в качестве параметра. Новый абзац при этом добавляется перед указанным блоком. В примере добавляется новый абзац перед вторым:

```
WD.Paragraphs.Add WD.Paragraphs(2).Range
```

Поиск и замена

Поиск и замена текста выполняются при помощи свойства **Find** объекта **Selection** (которое возвращает объект **Find**). В следующем примере показано использование объекта **Find** для поиска строки ABC. Комментарии поясняют параметры поиска:

```
Dim Result As Boolean
With WA.Selection.Find          ' Искать
    .Forward = True             ' в направлении "Вперед"
    .ClearFormatting            ' без учета форматирования
    .MatchWholeWord = False     ' искать слово целиком
    .MatchCase = False          ' без учета регистра
    .Wrap = wdFindStop          ' при нахождении остановиться
    Result = .Execute("ABC")    ' True, если найдено
End With
```

Свойство **Wrap** объекта **Find** управляет действиями поиска при обнаружении конца или начала выделения и может принимать значения:

wdFindStop = **0** — остановиться;
wdFindContinue = **1** — продолжить поиск с начала или с конца;
wdFindAsk = **2** — запросить пользователя о дальнейших действиях;

Замена одной подстроки другой выполняется аналогично, при этом заменяющая строка задается как параметр **ReplacementWith** метода **Execute** (десятый параметр метода).

Проверка правописания

Для проверки правильности написания слов *MS Word* использует словари. Слово сравнивается с имеющимися в словаре и, если слово в словаре не обнаружено, высказывается предположение о правильном написании.

Если слово не обнаружено в основном словаре, оно проверяется в дополнительных словарях (до десяти словарей можно указать в параметрах метода). Кроме проверки правописания *MS Word* проверяет также грамматику.

Проверить отдельное слово на правильность можно при помощи метода **CheckSpelling** объекта **Application**. Метод возвращает **False**, если слово не найдено в словаре, например:

```
If Not WA.CheckSpelling("Руму") Then
    ' неправильное слово
End If
```

Вычислить все предположения о возможном правильном написании слова можно при помощи метода **GetSpellingSuggestions**, который возвращает коллекцию предположений **SpellingSuggestions**. В следующем примере предполагается, что элементы коллекции заносятся в элемент управления **List1** типа **ListBox**:

```
If Not WA.CheckSpelling("Руму") Then
    Dim SS As SpellingSuggestions, S As SpellingSuggestion
    Set SS = WA.GetSpellingSuggestions("Руму")
    For Each S In SS
        List1.AddItem S.Name
    Next
End If
```

Наличие грамматических ошибок в тексте можно проверить при помощи метода **CheckGrammar** объекта **Application**. Метод возвращает **True**, если поданное на вход предложение не содержит (с точки зрения *MS Word*) грамматических ошибок:

```
If Not (WA.CheckGrammar("пример пример") Then
    ' грамматические ошибки
End If
```

Ограниченный объем данного учебного пособия не позволяет осветить все возможности по управлению приложением *MS Word*. Для дальнейшего изучения можно порекомендовать использовать запись макросов *MS Word*, чтобы посмотреть, как выполняются те или иные действия.

Проектирование сервера-приложения

Сервер автоматизации (сервер типа *ActiveX EXE*) может работать либо как *чистый сервер автоматизации*, либо как сервер-приложение (называемый в настоящем пособии автоматизированным приложением). Во втором случае сервер может выступать также в качестве чистого сервера автоматизации (просто сервера автоматизации). Сервер автоматизации отличается тем, что может быть использован в любых средах программирования, допускающих связывание через интерфейс *IDispatch*.

В этом разделе рассматриваются различные аспекты проектирования сервера-приложения автоматизации средствами системы программирования *MSVB*. Можно выделить следующие главные задачи, которые при этом нужно решить:

- разработать модель данных приложения;
- определить функции приложения;
- разработать объектную модель приложения;
- спроектировать интерфейс приложения.

Обозначенные задачи достаточно сильно взаимосвязаны, чтобы их можно было решить раздельно и независимо, поэтому проектирование автоматизированного приложения представляет собой итеративный процесс, как, впрочем, и разработка приложения любого другого вида или типа [3].

Модель данных определяет те элементы данных, которыми приложение управляет, и может быть определена на разных уровнях абстракции — логическом, функциональном, физическом. Разработка модели данных в наиболее полном объеме изучается в курсе «Теория разработки программного обеспечения». Эта часть проектирования приложения не является целью настоящего пособия и здесь не описывается. При необходимости можно обратиться, например, к учебнику [3]. Однако для разработки сервера-приложения важно четко определить используемые структуры данных и организовать программный интерфейс для доступа к ним. Более подробно эти вопросы освещаются в разделе «Сохранение данных».

Функции приложения определяют взаимодействие и преобразование данных (согласно [3], этот этап называется моделированием обработки). Как правило, функции приложения включают в себя добавление, удаление, поиск и модификацию элементов данных и обеспечивают реализацию так называемых бизнес-функций. В настоящем пособии функции приложения рассматриваются в контексте взаимосвязи интерфейса и объектной модели. Подробнее см. раздел «Функции приложения».

Проектирование интерфейса, являясь одной из важнейших задач при проектировании приложения, также выходит за рамки данного учебного курса. Подробнее с основами разработки интерфейса студенты знакомятся в курсе «Человеко-машинное взаимодействие». Тем не менее, ряд важных вопросов, касающихся организации и реализации интерфейса, освещается в разделе «Основы создания интерфейса приложения».

Разработка объектной модели является центральной частью данного курса, и поэтому данный раздел пособия достаточно подробно описывает различные проблемы, возникающие при решении этой задачи.

Базовая модель сервера автоматизации

После создания проекта типа *ActiveX EXE* прежде всего нужно открыть свойства проекта, выбрав в меню *Project—Project1 Properties*, и установить *название проекта* на вкладке *General* в поле *Project Name*. Это ответственный момент, поскольку для проектов типа *ActiveX* название проекта определяет *название сервера*, которое записывается в реестр и используется при создании ссылок на классы. Поэтому название сервера должно быть тщательно продумано, отражать цель проекта и по возможности содержать не более 8 знаков. Название проекта составляется из букв, цифр и знака подчеркивания (как идентификатор).

Далее на вкладке *Component* следует выбрать тип сервера: *Standalone* для сервера типа «приложение», или *ActiveX Component* для чистого сервера автоматизации.

Выбор типа сервера влияет на то, как создается объект автоматизации. Если сервер будет работать только как сервер, ссылку на объект автоматизации создает контроллер автоматизации. При этом сервер может содержать только классы автоматизации и, возможно, вспомогательные классы и модули.

Если же сервер предполагается использовать и как приложение, то ссылку на объект автоматизации должен создавать сам сервер. Для этого используется процедура **`Main`**, которую располагают в стандартном модуле проекта. Работа сервера начинается с функции **`Main`**. Внутри этой функции необходимо проверить режим старта, на который указывает свойство **`StartMode`** глобального объекта **`App`**. Это свойство может принимать два значения: **`vbSModeAutomation`** (режим сервера) и **`vbSModeStandalone`** (режим приложения).

Если сервер стартовал в режиме приложения, следует создать локальную переменную для объекта автоматизации и создать объект. Поскольку

локальная переменная прекратит свое существование по завершении процедуры **Main**, класс автоматизации должен сохранить ее, чтобы приложение не завершило свою работу сразу после старта. Так как приложение имеет основное окно, сохранить ссылку на класс автоматизации лучше всего в классе этого окна. Окно приложения следует загрузить в память. Загруженная в память форма окна будет удерживать ссылку на класс автоматизации до своего уничтожения.

Рассмотрим пример простейшего сервера автоматизации, состоящего из класса **Application**, формы основного окна **Form1** и модуля **Module1**, который содержит процедуру **Main**. Процедура **Main** создает объект автоматизации и делает основное окно приложения видимым:

```
Public Sub Main()
    On Error Resume Next
    If (App.StartMode = vbSModeAutomation) Then Exit Sub
    ' Старт в режиме приложения
    Dim A As New Application
    ' Показываем основное окно
    A.Visible = True
End Sub
```

Когда исполнение в этой процедуре доходит до строки

```
    ' Старт в режиме приложения
    Dim A As New Application
```

срабатывает конструктор класса **Application**. Класс **Application** содержит ссылку на форму основного окна, конструктор, деструктор и свойство **Visible**. Конструктор класса инициализирует ссылку на форму и загружает ее в память.

```
' Ссылка на форму основного окна
Private MF As Form1

Private Sub Class_Initialize()
    On Error Resume Next
    Set MF = New Form1
    ' Загружаем форму в память
    Load MF
    ' Сохраняем ссылку на класс автоматизации
    Set MF.Application = Me
End Sub
```

При выполнении строки кода конструктора

```
' Сохраняем ссылку на класс автоматизации
Set MF.Application = Me
```

ссылка на класс автоматизации дублируется при помощи свойства **Application** в объекте формы, которая к этому моменту загружена в память, но не отображается на экране (невидима).

Деструктор класса **Application** выгружает форму из памяти, что приводит к уничтожению удерживающей класс ссылки и дает возможность классу **Application** прекратить свое существование.

```
Private Sub Class_Terminate()
    On Error Resume Next
    Unload MF
End Sub
```

Далее в классе **Application** описывается свойство **Visible**, которое управляет видимостью приложения. Это свойство управляет аналогичным свойством формы.

```
' Возвращает/устанавливает признак видимости окна приложения.
Public Property Get Visible() As Boolean
    On Error Resume Next
    Visible = MF.Visible
End Property

Public Property Let Visible(ByVal NewValue As Boolean)
    On Error Resume Next
    MF.Visible = NewValue
End Property
```

Форма основного окна приложения содержит свойство **Application** для хранения ссылки на класс автоматизации:

```
' Ссылка на класс автоматизации
Private pApplication As Application

Public Property Get Application() As Application
    On Error Resume Next
    Set Application = pApplication
End Property

Friend Property Set Application(ByVal NewValue As Application)
    On Error Resume Next
    Set pApplication = NewValue
End Property
```

Объектная модель приложения

Существует три основных модели приложений *Windows*. Отличительным признаком той или иной модели является наличие или отсутствие документа или документов приложения. Документ приложения в терминологии *Windows* — данные, с которыми приложение работает, сохраняемые в виде файла определенного типа. Этот файл также называют документом. Так, например, документ приложения *Paint* — это файл типа `.bmp`, а документ приложения *Блокнот* — это файл типа `.txt`.

Приложение диалогового типа

Приложение диалогового типа состоит из одного основного окна приложения, *на котором располагаются элементы управления* для выполнения функций приложения. Примером такого приложения является калькулятор *Windows*. Отличительная особенность окна приложения — оно имеет неизменный размер, как и обычное диалоговое окно.

Как правило, такие приложения используются в случае, если отсутствует документ приложения. Функциональность приложения в этом случае реализуется элементами управления в основном окне а также при помощи меню приложения.

Приложение типа SDI

SDI расшифровывается как *Single Document Interface* (однодокументный интерфейс). Это означает, что в один момент времени приложение работает с данными только одного документа. Примерами таких приложений являются приложения *Windows Блокнот* и *Paint*.

Основное окно приложения может изменять свой размер и *содержит в своей основной части* рабочей области *содержимое документа*, например, текст или рисунок. Функции приложения при этом инициируются при помощи меню и панелей инструментов.

Приложение типа MDI

MDI расшифровывается как *Multiple Document Interface* (многодокументный интерфейс). Это означает, что в один момент времени приложение может работать с данными нескольких документов. Примерами таких приложений являются приложения *Microsoft Office*.

Основное окно приложения типа *MDI* является контейнером для окон документов (которые располагаются внутри основного окна, имеющего специальный тип MDI).

Соответствие объектной модели типу приложения

Объектная модель приложения — это отражение его реальных видимых и невидимых объектов, поэтому она должна в точности соответствовать модели приложения.

Приложение диалогового типа

Объектная модель приложения диалогового типа может состоять из одного единственного класса — **`Application`**. При этом все функции приложения реализуются как свойства и методы этого класса (рисунок 9):

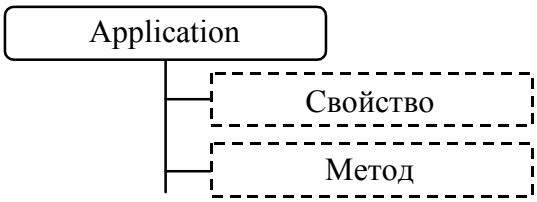

Рисунок 9 — Базовая объектная модель приложения диалогового типа

Приложение типа SDI

Приложение типа *SDI* работает с документом, поэтому объектная модель приложения состоит как минимум из класса **`Application`** и класса **`Document`**. Класс **`Application`** выполняет функции, присущие приложению в целом, а класс **`Document`** — функции работы с документом (рисунок 10).

С классом приложения связывается основное окно приложения, а с классом документа — элементы управления, отвечающие за отображение информации документа на экране. В связи с тем, что с классом документа не связано никакое окно, класс документа в некоторых случаях может отсутствовать. Это возможно в случае, если документ имеет достаточно простую структуру.

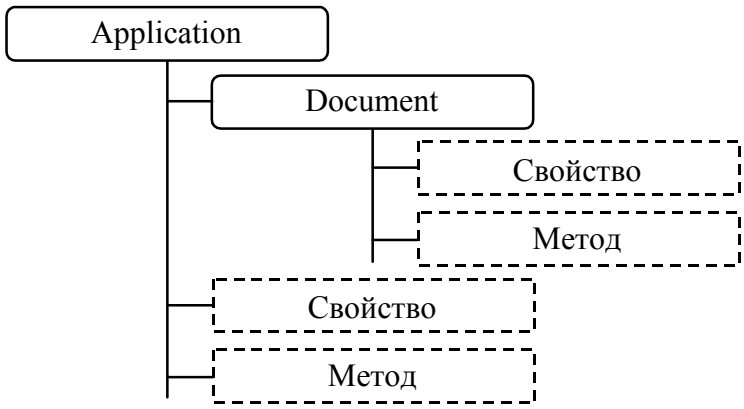

Рисунок 10 — Базовая объектная модель приложения типа SDI

Это не единственно возможная модель. В некоторых случаях класс документа является элементом коллекции документов, которая при этом может содержать только один документ, который создается автоматически. Так, например, устроена объектная модель приложения AutoCAD версии 15. Такое построение объектной модели, более характерное для приложения типа *MDI*, позволяет лучше структурировать функциональные элементы.

Приложение типа MDI

Приложение типа *MDI* является наиболее сложным. Объектная модель такого приложения состоит из трех базовых классов (рисунок 11): класс **`Application`** отвечает за функции приложения в целом, класс **`Documents`** представляет собой коллекцию всех открытых документов, класс **`Document`** представляет данные одного из документов.

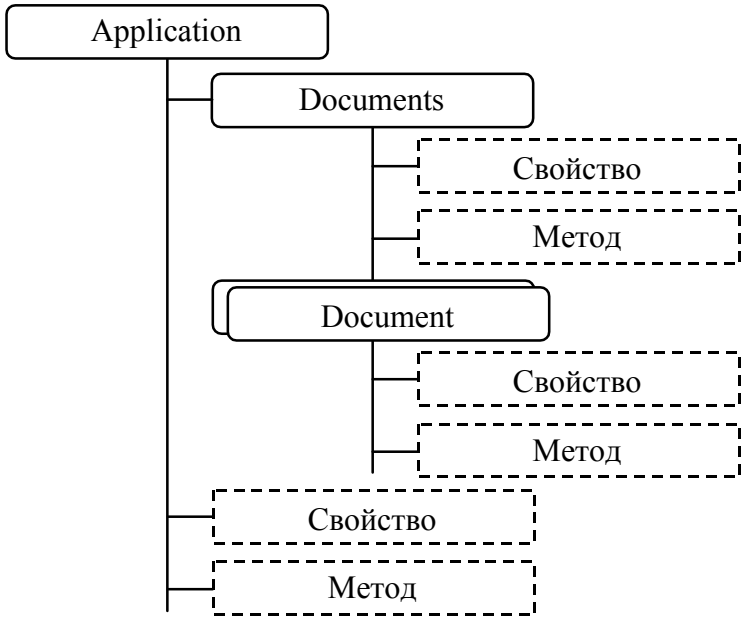

Рисунок 11 — Базовая объектная модель приложения типа MDI

Важное отличие данной объектной модели от предыдущей — наличие коллекции **`Documents`**, которая обладает свойствами и методами, специфичными для коллекций, и дополнительно может иметь свойства и методы, специфичные для приложения (или объекта коллекции).

Классы объектной модели

Для создания объектной модели приложения используются классы разных типов и назначения. При проектировании классов в среде *MSVB* каждому классу устанавливается свойство **`Instancing`**, которое определяет,

каким образом данный класс будет использоваться внутри и вне объектной модели.

Свойство Instancing

Свойство **Instancing** может принимать следующие значения:

1 — **Private**. Представитель данного класса не может быть создан вне объектной модели и является внутренним и невидимым. Пользователь не может создавать ссылки на представителя данного класса.

2 — **PublicNotCreatable**. Представитель данного класса не может быть создан вне объектной модели, однако является видимым. Пользователь может создавать ссылки на представителя данного класса и использовать свойства и методы полученного представителя через ссылку. Представитель класса при этом создается внутри объектной модели (сервера).

3 — **SingleUse**. Представитель данного класса может быть создан пользователем при помощи генератора **New** или функции **CreateObject**. Каждый созданный представитель класса приводит к созданию копии компонента (сервера), который его содержит. Данный тип класса недопустим для серверов типа ActiveX DLL.

4 — **GlobalSingleUse**. Аналогично значению **SingleUse**, за исключением того, что для использования свойств и методов представителя данного класса вне объектной модели (вне сервера) не требует создания представителя класса. Свойства и методы становятся глобальными в приложении клиента (контроллера автоматизации). Данный тип класса недопустим для серверов типа ActiveX DLL.

5 — **MultiUse**. Аналогично значению **SingleUse**, однако при создании нового представителя класса не создается новая копия компонента (сервера). В некоторых случаях это может привести к невозможности создания второго представителя данного класса.

6 — **GlobalMultiUse**. Аналогично значению **MultiUse**, но для использования свойств и методов представителя класса вне объектной модели (вне сервера) не требуется создания представителя класса.

Группы классов

Классы объектной модели можно поделить на следующие группы:

- классы, представляющие сущности;
- классы, представляющие коллекции сущностей;
- глобальные классы;
- вспомогательные классы.

Классы, представляющие *сущности*, описывают свойства и методы действующих (активных) объектов программы. Свойства описывают характеристики или атрибуты объектов, методы описывают их поведение.

Например, если приложение работает с документом, то свойства соответствующего класса описывают элементы документа, такие, как фамилия автора, текст, дата создания и т.п., а методы — действия, такие как сохранение или чтение документа.

В свою очередь, элементы документа также могут быть сущностями и описываться при помощи дополнительных классов. При этом свойства класса документа могут иметь объектный тип, возвращающий соответствующий объект.

Классы, представляющие *коллекции сущностей*, предназначены для перечисления объединяемых сущностей. Для этой цели в этих классах создается *объект перечисления*, при помощи которого можно выполнять последовательный их перебор при помощи, например, оператора **`For Each`** (в языке *MSVB*).

Кроме того, классы коллекций обладают общими методами и свойствами, предназначенными для создания новых элементов коллекций, их удаления, выбора произвольного элемента на основе его порядкового номера или уникальной ключевой строки, а также определения количества элементов коллекции. В силу сказанного классы коллекций строятся одинаковым образом и чаще всего получаются из имеющейся коллекции простым копированием файла и последующей заменой имен типов.

Глобальные классы используются без образования ссылки на своего представителя (вне сервера). Как правило, это классы, определяющие константы, перечисления, типы и функции, имеющие общее значение. Для создания глобального класса необходимо установить свойство **`Instancing`** равным 4 (**`GlobalSingleUse`**) или 6 (**`GlobalMultiUse`**).

Целочисленные константы обычно задаются в виде общего перечисления, тип которого на самом деле нигде не используется, например:

```
Public Enum XXXXGeneral
    [Название Константы] = целочисленное_значение
End Enum
```

Вещественные и строковые константы могут быть заданы только при помощи свойств соответствующего типа, например:

```
Public Property Get csCompanyName() As String
    csCompanyName = "Название организации"
End Property
```

Примерами функций, которые могут использоваться повсеместно в приложении, являются функции пересчета размерных величин из одной единицы измерения в другую, функции различных математических методов, используемых программой, другие функции, для вычисления значений которых не требуются представители базовых классов.

Вспомогательные классы предназначены для описания поведения, которое является внутренним по отношению к пользователю. Вспомогательные классы имеют свойство **`Instancing`** равное **`1`** — **`Private`**, поэтому они не видны пользователю и отсутствуют в объектной модели (не экспортируются ею). Если для хранения данных приложения используется, например, база данных, то для выполнения операций с базой данных может быть использован один или несколько вспомогательных классов.

Построение коллекции классов

Коллекции отличаются от обычных классов тем, что они не представляют никакие действующие (активные) объекты приложения, но позволяют одинаковым образом управлять множествами однотипных таких объектов. Коллекции — чрезвычайно удобное и мощное средство объектного описания поведения программы и отдельных ее частей.

Как правило, коллекция обладает следующим стандартным набором свойств и методов:

1. Метод **`Add`** (функция) предназначен для добавления нового элемента в коллекцию. Метод возвращает ссылку на добавленный элемент.

2. Свойство **`Count`** возвращает количество элементов в коллекции.

3. Свойство **`Item`** возвращает элемент коллекции по его порядковому номеру или по ключевому слову (фразе), в зависимости от типа значения передаваемого методу параметра. Если параметр числовой, подразумевается порядковый номер элемента коллекции. Если параметр имеет строковый тип, подразумевается ключевое слово. Ключевое слово задается при включении элемента в коллекцию (в методе **`Add`**). Свойство **`Item`** является свойством по умолчанию.

4. Метод **`Remove`** (процедура). Удаляет из коллекции элемент по его порядковому номеру или по ключевому слову.

При построении коллекции необходимо также позаботиться о том, чтобы она формировала объект перечисления, необходимый для работы циклической конструкции **`For Each`** в языках типа *Visual Basic*.

Построение класса коллекции в *MSVB* производится на основе класса **`Collection`**, который обладает всеми перечисленными свойствами и мето-

дами. Цель построения класса — определить уникальную коллекцию (работающую с классами определенного типа).

На уровне модуля в классе коллекции объявляется следующая закрытая переменная (базовая коллекция):

```
Private pColl As Collection
```

В конструкторе класса переменная `pColl` инициализируется:

```
Private Sub Class_Initialize()
    Set pColl = New Collection
End Sub
```

В деструкторе класса переменная `pColl` очищается:

```
Private Sub Class_Terminate()
    Set pColl = Nothing
End Sub
```

Метод Add

Простейшая форма метода `Add` заключается в создании нового представителя типа коллекции (обозначенного в примерах как `ТИП_КОЛЛЕКЦИИ`) и включении его в коллекцию при помощи метода `Add` класса `Collection`:

```
Public Function Add() As ТИП_КОЛЛЕКЦИИ
    Dim Q As New ТИП_КОЛЛЕКЦИИ
    pColl.Add Q
    Set Add = Q
End Function
```

Свойства такой формы создания элемента коллекции следующие:

- начальные свойства вновь созданного представителя класса задаются конструктором класса;
- элемент коллекции добавляется в конец коллекции;
- элемент коллекции индексируется только порядковым номером, так как при добавлении элемента в `pColl` не задается ключ.

Дальнейшие действия с вновь созданным таким образом представителем класса обычно подразумевают установку его специфических значений свойств, например, так:

```
With Коллекция.Add
    .Свойство = Новое_Значение
End With
```

На практике часто может оказаться неудобным использовать простую форму добавления нового элемента в коллекцию. Возможны следующие модификации метода `Add`:

Включение элемента в определенное место в коллекции

Чтобы включить элемент коллекции в определенное место, следует использовать либо третий, либо четвертый параметр метода **Add** класса **Collection** (но не оба вместе). Третий параметр (**Before**) используется, когда новый элемент вставляется перед существующим элементом коллекции. Четвертый параметр (**After**) используется, когда новый элемент вставляется за существующим элементом коллекции. В качестве параметров используется либо порядковый номер, либо уникальный строковый ключ элемента коллекции. В следующем примере показано, как модифицировать метод **Add** для добавления нового элемента в начало коллекции:

```
pColl.Add Q, , 1
```

Заметим, что если указываемый для ориентации элемент в коллекции отсутствует, возникнет ошибка и новый элемент не будет добавлен в коллекцию. Поэтому перед выполнением данной строчки кода обязательно нужно убедиться, что коллекция не пуста, например, так:

```
If (pColl.Count = 0) Then
    pColl.Add Q
Else
    pColl.Add Q, , 1
End If
```

С помощью указанной техники можно легко организовать коллекцию таким образом, что она будет упорядоченной по некоторому признаку включаемых в нее элементов.

Определение ключа элемента коллекции

Неизвестно почему, но, как показывает практика, ключ элемента коллекции вызывает стойкую неприязнь у всех впервые сталкивающихся с коллекцией программистов. На самом деле коллекция с ключами представляет собой ассоциативный массив. Ключи либо задаются для всех элементов коллекции, либо не задаются ни для каких элементов. Если ключи задаются, то они должны быть уникальными для всех элементов коллекции. Ключ используется как для получения элемента коллекции при помощи свойства **Item**, так и для удаления элемента коллекции при помощи метода **Remove**. Параметр **Index** этих процедур имеет тип **Variant**, что дает возможность указывать как порядковый номер, так и ключ.

Ключ задается вторым параметром метода **Add** класса **Collection**. И самым сложным является выбор ключа. Простейший способ создать ключ — использовать порядковый номер, например:

```
pColl.Add Q, CStr(pColl.Count + 1)
```

Здесь новый элемент получает ключ, равный значению числа элементов коллекции плюс один — под таким порядковым номером новый элемент окажется в коллекции. В этом ключе нет особого смысла, однако его легко модернизировать, например, так:

```
pColl.Add Q, "Документ" & CStr(pColl.Count + 1)
```

Здесь ключом является слово с номером добавленного элемента. Этот способ мало чем отличается от предыдущего, однако если вместо номера элемента коллекции использовать, например, номер окна, связанного с данным объектом, то впоследствии это даст возможность легко вычислить ключ, используя название окна. Таким образом, ключ должен быть таким, чтобы его можно было вычислить позднее.

Еще один из вариантов ключа — использовать специальное свойство включаемого объекта, которое предоставляет уникальный ключ каким-либо способом. В следующем примере в качестве ключа используется идентификатор записи в базе данных, с которой включаемый объект связывается (**БАЗА_ДАННЫХ** — ссылка на класс базы данных):

```
Public Function Add() As ТИП_КОЛЛЕКЦИИ
    Dim Q As New ТИП_КОЛЛЕКЦИИ, ID As Long
    ID = БАЗА_ДАННЫХ.ObjectAddNew
    If (ID = 0) Then Err.Raise cerrObjectAddNewFailure
    Q.ID = ID
    pColl.Add Q, CStr(ID)
    Set Add = Q
End Function
```

Так как идентификатор объекта всегда можно узнать впоследствии, ключ всегда может быть вычислен. Не следует однако забывать, что если идентификатор не доступен пользователю объектной модели (например, контроллеру), то и ключ также будет недоступен.

Дополнительно о ключах коллекции см. приложение «Занятия», описываемое в разделе «Построение объектной модели».

Формирование нового объекта вне метода Add

Иногда полезно (или удобно) сформировать новый объект вне метода **Add**, а затем передать его методу **Add** как параметр. Метод **Add** при этом может иметь следующий простой вид:

```
Public Function Add(ByVal NewItem As ТИП_КОЛЛЕКЦИИ) As ТИП_КОЛЛЕКЦИИ
    pColl.Add NewItem
    Set Add = NewItem
End Function
```

Достоинство этого способа — свойства нового объекта могут быть установлены заранее. Такая необходимость может возникнуть, например, при добавлении нового объекта, связанного с записью в базе данных. Если при добавлении новой записи требуется заполнить определенные ее поля определенными значениями (например, связать с записями в других таблицах), это потребует знать значения этих полей до добавления объекта, а не после этого. На самом деле эта проблема может быть решена множеством способов. Самый простой — использовать значения по умолчанию в полях, требующих обязательных значений в базе данных.

Недостаток описываемого способа формирования метода **Add** очень серьезный. Для того, чтобы добавить новый объект, сначала его нужно создать. Это легко можно сделать внутри сервера. Но сделать это вне сервера может оказаться невозможным ввиду значения свойства **Instancing**, установленного для данного класса. На самом деле только класс **Application** в большинстве практических случаев является классом, который может быть создан вне сервера. Другие классы вне сервера создавать нельзя даже не потому, что в этом нет особого смысла, а потому, что все классы внутри сервера связаны друг с другом некоторым образом, и использование представителя класса вне его связи с другими объектами объектной модели может привести к разрушению приложения.

Формирование параметров метода Add

Выходом из ситуации, описанной в предыдущем пункте, может быть решение использовать некоторое количество параметров в методе **Add**, которые совершенно необходимы для задания значений при создании нового элемента коллекции. Если такие значения есть, метод **Add** может принимать и следующую форму:

```
Public Function Add(ByVal Key As String, ByVal Value As String) As
ТИП_КОЛЛЕКЦИИ
    Dim Q As New ТИП_КОЛЛЕКЦИИ
    ' задаются свойства нового объекта
    Q.Value = Value
    pColl.Add Q, Key
    Set Add = Q
End Function
```

Здесь параметрами метода являются ключ и какое-то значение, которое новый объект получит как значение свойства во время своего создания. Другие свойства объекта могут быть установлены позднее.

Добавление разнотипных объектов

Иногда возникает необходимость в добавлении в коллекцию объектов разных типов. Например, коллекция элементов какой-нибудь схемы может состоять из элементов схемы, которые по разным причинам не могут быть описаны одним классом. В принципе, класс **Collection** позволяет объединять совершенно произвольные объекты. Но при конструировании класса разнородной коллекции возникают разные конфликтные ситуации. Например, если новый объект генерируется методом **Add**, то нужен способ указания типа нового объекта. Кроме того, если метод **Add** возвращает новый элемент, то следует решить, какой тип метод должен иметь.

Есть два практических способа реализации коллекции разнотипных объектов. Первый заключается в том, чтобы определить общий интерфейс для всех включаемых в коллекцию типов объектов, и использовать в качестве элементов коллекции ссылки на этот интерфейс. Второй способ заключается в разработке множественных методов **AddXXXX**, каждый из которых предназначен для включения в коллекцию объектов одного из типов. Не возникнет противоречия при использовании обоих подходов одновременно. Так, например, разработана коллекция элементов пространства модели в приложении AutoCAD.

Свойства Count и Item

Реализация свойств **Count и Item** тривиальна, — используются свойство базового класса **Collection**:

```
Public Property Get Count() As Integer
    Count = pColl.Count
End Property

Public Property Get Item(ByVal Index As Variant) As ТИП_КОЛЛЕКЦИИ
    Set Item = pColl.Item(Index)
End Property
```

После описания свойства **Item** нужно установить атрибут процедуры **DISPID_VALUE** (**Default**), иначе оно не будет свойством по умолчанию.

Метод Remove

Реализация метода **Remove** также тривиальна, однако в некоторых случаях в этой процедуре выполняются дополнительные действия, которые могут привести к отмене удаления элемента коллекции.

Как правило, отмена удаления возникает, если нельзя удалить элемент, связанный с записью в базе данных (вследствие существующего в

ней ограничения целостности). В этом случае примерный вид процедуры удаления элемента коллекции может быть таким:

```
Public Sub Remove(ByVal Index As Variant)
    Dim Q As ТИП_КОЛЛЕКЦИИ
    Set Q = pColl.Item(Index)
    If Not БАЗА_ДАННЫХ.ObjectRemove(Q.ID) Then Exit Sub
    pColl.Remove Index
End Sub
```

Как и в методе **Add**, здесь **БАЗА_ДАННЫХ** — это ссылка на класс базы данных.

Метод NewEnum

Коллекция будет полностью определена в случае, если для нее реализован метод (или свойство), возвращающий так называемый объект перечисления. При создании коллекции средствами *MSVB* этот метод реализуется следующим стандартным способом:

```
Public Function NewEnum() As IUnknown
    Set NewEnum = pColl.[_NewEnum]
End Function
```

Данный метод будет работать, только если определен атрибут процедуры **DISPID_NEWENUM**, равный **–4**.

Основные ошибки проектирования коллекции

После создания коллекции ее необходимо протестировать, чтобы убедиться, что она правильно работает. Пусть проектируемая коллекция имеет простейший вид. Тогда должны работать следующие программные конструкции:

1) Добавление нового элемента

```
With КОЛЛЕКЦИЯ.Add
    .Value = значение
End With
```

Если возникает ошибка при выполнении первой строки, метод **Add** не возвращает объектный тип (в описании метода).

Если возникает ошибка во второй строке (которая приведена здесь как пример кода, а не как действительный код), метод **Add** либо не создает новый объект, либо возвращает неправильный тип.

2) Свойство *Item*

```
Dim Q As ТИП_КОЛЛЕКЦИИ
Set Q = КОЛЛЕКЦИЯ(1)
```

Если во второй строке возникает ошибка, метод **Item** либо отсутствует, либо для него не установлен атрибут **DISPID_VALUE** (**Default**).

3) Метод (свойство) *NewEnum*

```
Dim Q As ТИП_КОЛЛЕКЦИИ
For Each Q In КОЛЛЕКЦИЯ
    Debug.Print Q.Value
Next
```

Если данный код во второй строке приводит к ошибке, метод **NewEnum** либо не определен, либо определен неправильно, либо для него не установлен атрибут **DISPID_NEWENUM**, равный –4. Здесь в цикле приведен примерный код. Действительный код зависит от типа элемента коллекции.

Во всех примерах **КОЛЛЕКЦИЯ** — это ссылка на действительный представитель класса коллекции.

Дополнительные методы коллекций

Метод Clear

Метод **Clear** используется для очистки коллекции для ее повторного заполнения. Его реализация достаточно тривиальна:

```
Public Sub Clear()
    Dim I As Long, N As Long
    On Error Resume Next
    N = pColl.Count
    For I = 1 To N
        pColl.Remove 1
    Next
End Sub
```

Метод Show

Часто коллекции связаны с диалоговым окном приложения. Диалоговое окно предназначено для управления элементами коллекции на уровне интерфейса пользователя. Каждое окно приложения должно быть связано с классом, который им управляет, поскольку прямое управление окном из объектной модели невозможно — класс окна является закрытым.

Для отображения окна на экране коллекция использует метод **Show**.

Стандартный вид метода следующий:

```
Public Sub Show()
    Dim Q As New frmXXXX
    Load Q
    With Q
        Set .Parent = Me
        .DisplayItems
        .Show vbModal
    End With
    Unload Q
End Sub
```

Здесь метод формы **DisplayItems** является процедурой отображения элементов коллекции в элементах управления окна. Этот метод использует внутреннюю ссылку на класс коллекции **pParent**, которая устанавливается после загрузки формы в память.

Метод Open

Метод **Open** используется в случаях, когда элемент коллекции добавляется в коллекцию при чтении его свойств из какого-либо хранилища данных. Параметром метода может быть либо путь к файлу хранилища, либо название элемента, используемое для его идентификации в стандартном хранилище. Чтение свойств элемента при этом чаще всего предполагает одновременное их отображение в графическом интерфейсе приложения. Здесь можно дать лишь самые общие рекомендации.

Если свойства элемента коллекции отображаются в интерфейсе приложения, то должно быть установлено соответствие между некоторым окном приложения и элементом коллекции. Рекомендуется в форме окна создать метод типа **DisplayItem**, который отвечает за вывод элемента в окно, и вызвать этот метод после завершения чтения свойств элемента.

```
Public Sub Open(ByVal Path As String)
    Dim Q As New ТИП
    With Q
        ' Прочитать объект из файла и установить свойства
    End With
    Dim I As Integer
    ' Добавить элемент в коллекцию
    I = pColl.Count + 1
    pColl.Add Q, CStr(I)
    ' Вывести свойства в окно
    pApplication.Window.DisplayItem Q
End Sub
```

Построение объектной модели

На самом верхнем уровне иерархии всегда находится класс приложения **Application**. Он предоставляет *все функции* управления приложением через свои свойства и методы. При этом сам класс реализует свойства и методы, отвечающие за работу приложения в целом, а другие функции приложения реализуют классы, ссылки на которые могут быть получены через объектные свойства класса **Application**. В связи с этим класс самого приложения должен иметь свойство **Instancing**, равное 3 — **SingleUse** или 5 — **MultiUse**, а другие классы приложения — свойство **Instancing**, равное 2 — **PublicNotCreatable**. Если, например, приложение реализует объектную модель, приведенную на рисунке 10, то доступ к документу приложения осуществляется при помощи следующей объектной записи:

```
Dim D As Document
Set D = A.Document
```

Если же приложение реализует объектную модель, приведенную на рисунке 11, то доступ к первому документу осуществляется при помощи следующей записи:

```
Dim D As Document
Set D = A.Documents(1)
```

Во многих практических случаях объектная модель намного сложнее, что приводит к более сложным взаимоотношениям между классами. В качестве примера будем рассматривать учебное приложение под названием **RECIPEE** — сборник кулинарных рецептов.

Основными классами приложения **RECIPEE** являются **Application** и **Recipe**. Класс **Recipe** (рецепт) представляет описание рецепта, состоящее из названия рецепта **Name**, описания способа приготовления **Description**, а также список компонентов с указанием их количества и единиц измерения. В связи с этим требуется новый класс **Component**, описывающий отдельный компонент рецепта, а для описания единицы его измерения — новый класс **Unit**. Все компоненты рецепта образуют коллекцию **Components**.

Далее, поскольку приложение представляет сборник рецептов, классы **Recipe** объединяются в коллекцию **Recipes**, которую формирует класс приложения **Application**. Для описания всех возможных ингредиентов используется класс **Ingredient**, а все ингредиенты образуют коллекцию (справочник) **Ingredients**. Аналогично класс **Application** формирует коллекцию всех возможных единиц измерения **Units**. В результате получается объектная модель, приведенная на рисунке 12.

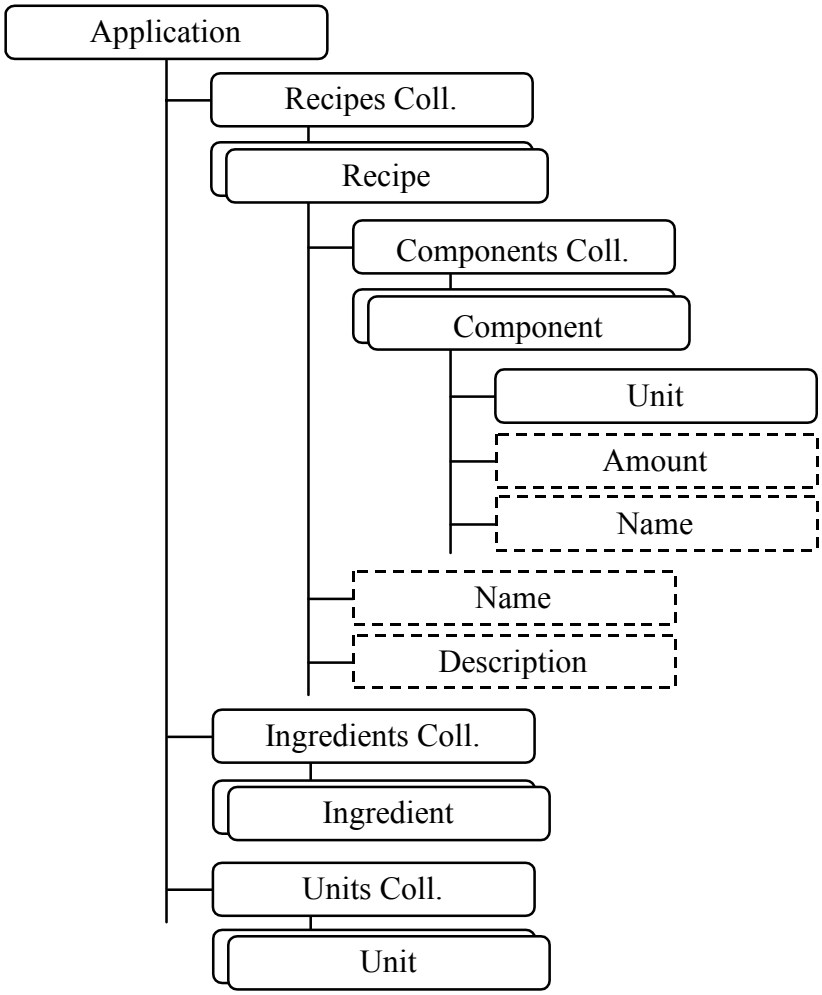

Рисунок 12 — Объектная модель приложения «Рецепты»

Этой объектной модели соответствуют следующие объектные записи (переменная **A** является ссылкой на класс **Application**):

- добавление новой единицы измерения в коллекцию единиц:

```
A.Units.Add "Название_длинное", "Название_короткое"
```

- добавление нового ингредиента в коллекцию ингредиентов:

```
A.Ingredients.Add "Название_ингредиента"
```

- добавление нового рецепта:

```
A.Recipes.Add
```

- добавление компонента в рецепт 1:

```
A.Recipes(1).Components.Add A.Ingredients("Название_ингредиента")
```

- установка количества компонента 1 рецепта 1:

```
A.Recipes(1).Components(1).Amount = Количество
```

- установка единицы измерения компонента 1 рецепта 1:

```
Set A.Recipes(1).Components(1).Unit = A.Units("Название_короткое")
```

Эти записи требуют осмысления. Они показывают, как на самом деле должно работать данное приложение. Например, что происходит, когда добавляется новый компонент в рецепт? Как показывает запись

```
A.Recipes(1).Components.Add A.Ingredients("Название_ингредиента")
```

в метод **Add** коллекции ингредиентов рецепта передается ссылка на элемент коллекции ингредиентов приложения. Такой подход гарантирует, что в коллекцию компонентов будет добавлен существующий ингредиент. При необходимости в методе **Add** делается проверка на действительность ссылки на ингредиент, например, так (**P** — аргумент метода **Add**):

```
If (P Is Nothing) Then Exit Function
```

Метод **Add** можно реализовать иначе. Вместо того, чтобы передавать объект, можно передавать название ингредиента, при этом объектная запись будет иметь следующий вид:

```
A.Recipes(1).Ingredients.Add "Название_ингредиента"
```

В этом случае метод **Add** сам выясняет, является ли данный ингредиент действительным ингредиентом коллекции ингредиентов приложения. Наконец, способ добавления ингредиента в коллекцию ингредиентов без указания какого-либо параметра представляется бесполезным.

При составлении объектной модели может оказаться возможным использовать один и тот же класс и одну и ту же коллекцию в разных контекстах. В случае с приложением **RECIPEE** классы **Component** и **Components** можно было бы реализовать как классы **Ingredient** и **Ingredients**. В этом случае в методах коллекции и класса коллекции потребовалось бы дополнительно проверять, какой класс является родительским, и в зависимости от этого реализовывать ту или иную функциональность класса. Для проверки типа родительского класса можно использовать следующий код:

```
If (TypeOf Объект Is Тип) Then . . .
```

Из приведенных выше объектных записей следует также, что коллекции **Recipes**, **Ingredients** и **Units** предоставляются классом **Application**. Этот класс отвечает за создание и экспорт коллекций. Для этого в нем создаются закрытые элементы:

```
' Коллекция рецептов
Private pRecipes As Recipes
' Коллекция единиц измерения
Private pUnits As Units
' Коллекция ингредиентов
Private pIngredients As Ingredients
```

В конструкторе класса **Application** коллекции создаются и инициализируются (в примере — коллекция **Units**):

```
Set pUnits = New Units
With pUnits
    Set .Application = Me
    Set .Parent = Me
    .InitFromDB pDatabase
End With
```

Как видим, сначала создается объект соответствующего класса, затем устанавливаются объектные ссылки и далее объект инициализируется значениями, считываемыми из базы данных.

Коллекция экспортируется классом **Application** при помощи свойства только для чтения. Например, для коллекции **Units** определяется свойство **Units**:

```
Public Property Get Units() As Units
    On Error Resume Next
    Set Units = pUnits
End Property
```

Часть свойства, устанавливающая новое значение, в данном контексте не имеет смысла, хотя в отдельных случаях это возможно. Например, в классе **Component** свойство, отвечающее за единицу измерения, является двунаправленным (вторая часть свойства имеет спецификатор **Set**):

```
Public Property Get Unit() As Unit
    On Error Resume Next
    Set Unit = pUnit
End Property

Public Property Set Unit(NewValue As Unit)
    On Error Resume Next
    Set pUnit = NewValue
End Property
```

Здесь **pUnit** — закрытая переменная класса **Component**.

Объектная модель должна отражать понятия предметной области, используемые пользователем. В качестве примера рассмотрим приложение «Занятия», управляющее расписанием занятий учебного заведения.

Занятие (*Lesson*) назначается на определенное время (*Time*) определенного дня (*Day*) определенной недели (*Week*), определенной группе (*Group*) в определенной аудитории (*Room*), на определенный предмет (*Subject*), определенному преподавателю (*Teacher*). Таким образом, в управле-

нии занятиями должны участвовать все перечисленные объекты, образующие объектную модель, приведенную на рисунке 13.

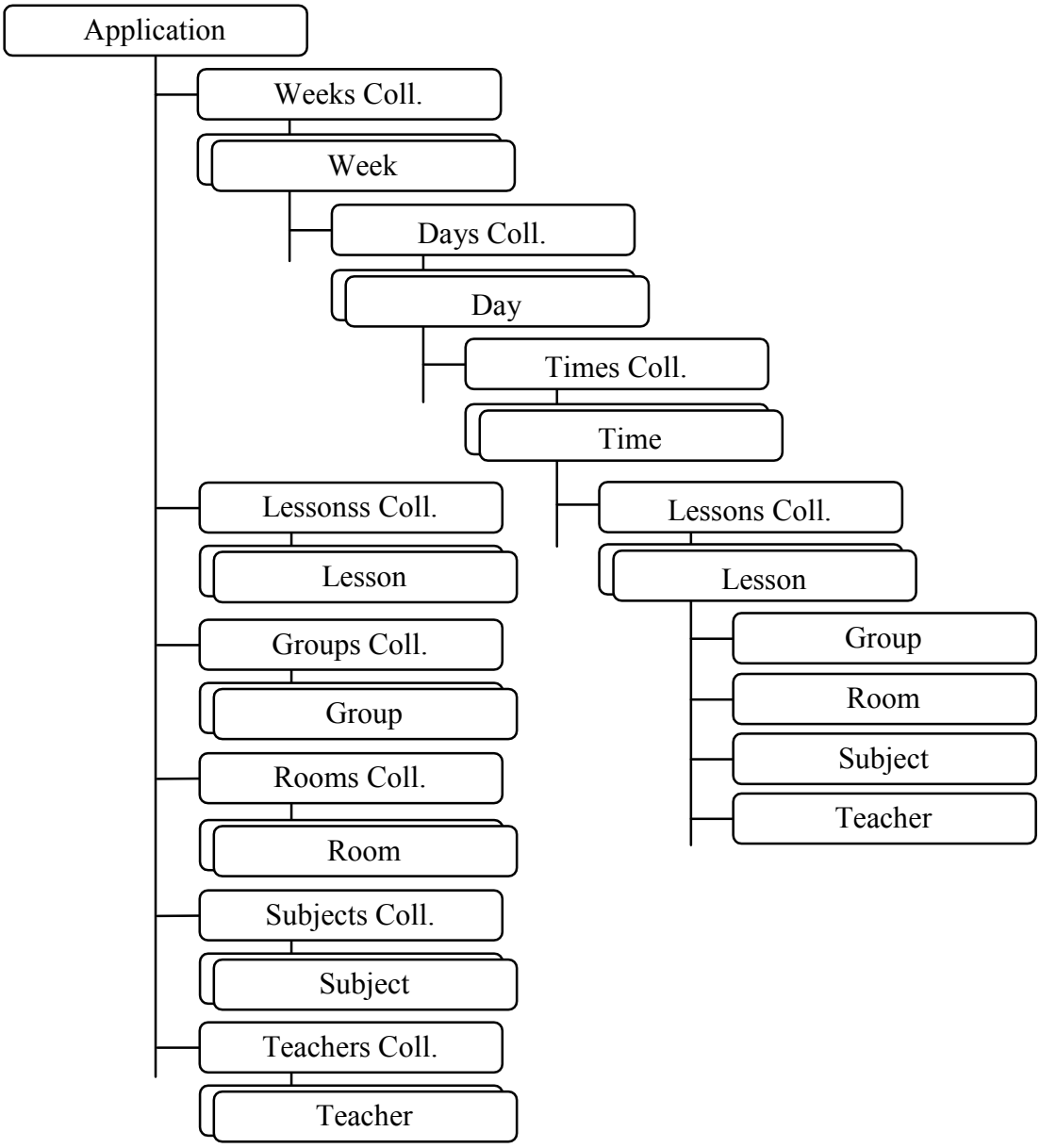

Рисунок 13 — Объектная модель приложения «Занятия»

Этой объектной модели соответствуют объектные записи, приведенные далее (переменная **A** — ссылка на класс приложения).

Добавление новой группы:

```
With A.Groups.Add("Название-ключ")
    Set .Application = A
    ' Другие свойства
End With
```

Аналогичным образом добавляются объекты других классов, за исключением занятия. Добавление нового занятия происходит примерно так:

```
Dim L As Lesson
Set L = A.Weeks(1).Days(1).Times(1).Lessons.Add(A.Groups(1))
```

Эта запись гарантирует, что определенной группе будет назначено занятие на определенной неделе в определенные день и время. При этом может быть сформирован простой ключ коллекции **Lessons**, состоящий из названия группы. Назначение аудитории, предмета и преподавателя осуществляется при помощи следующих объектных записей:

```
Set L.Room = A.Rooms(1)
Set L.Subject = A.Subjects(1)
Set L.Teacher = A.Teachers(1)
```

Заметим, что элементы коллекций в приведенных выше записях для простоты идентифицируются порядковыми номерами. На самом деле удобнее идентифицировать элементы коллекций ключевыми значениями, в качестве которых здесь могут выступать названия групп, аудиторий, предметов и, возможно, специальные идентификаторы для преподавателей и дней недели. Учебная неделя и время занятия могут быть идентифицированы порядковыми номерами.

При изменении группы занятия в предложенной схеме элемент коллекции удаляется и формируется заново. Заметим, что при этом возможно назначить одну и ту же группу в разные аудитории на одно и то же время (при, например, распараллеливании занятий по подгруппам).

При данном подходе к построению объектной модели возможно назначить одного и того же преподавателя в одно и то же время в разные аудитории. Чтобы исключить эту возможность, можно указывать в качестве параметров метода **Add** коллекции **Lessons** три параметра — группу, аудиторию и преподавателя, с формированием сложного составного ключа.

В данной объектной модели используются формальные классы недель **Week**, дней недели **Day** и времен занятий **Time** с соответствующими коллекциями. Классы называются здесь формальными, поскольку они не несут никакой функциональности, кроме обеспечения правильной взаимосвязи объектов. Фактически данные классы предназначены для идентификации учебной недели, дня и времени занятия, и могут не содержат никаких свойств и методов, за исключением, возможно, свойства типа **ID**, являющимся идентификатором объекта. Тем не менее, объект типа **Time** может экспортировать свойства, описывающие начало и конец занятия, объект

типа **Day** — фактическую дату, а объект типа **Weeks** — начало и конец учебного периода (семестра).

Объектная модель данного приложения может быть построена иначе. Предположим, коллекции всех классов принадлежат классу приложения. В этом случае добавление нового занятия будет включать в себя установку сразу 4-6 объектных ссылок, передаваемых методу **Add** коллекции **Lessons**. В такой модели способ выбора элемента коллекции **Lessons** является нетривиальным. Пользователь-программист будет вынужден составлять более сложные составные ключи для того, чтобы однозначно идентифицировать занятие. Кроме того, структура приложения в этом случае скрывает взаимоотношения реальных объектов, с которыми пользователю приходится иметь дело.

Класс Application

В зависимости от объектной модели, класс приложения может предоставлять различную функциональность, кроме описанной в предыдущих разделах. Если объектная модель отражает приложение диалогового типа (рисунок 9), то класс приложения реализует методы, связанные с сохранением и чтения элементов данных при помощи методов типа **XXXXSave** и **XXXXOpen**, где **XXXX** это название объекта сохранения (например, **Document**, **Data**, **List**). Если объектная модель отражает приложение с одним документом (рисунок 10), класс приложения экспортирует метод **XXXXOpen**, с помощью которого документ можно открыть. Одновременно класс приложения может предоставлять возможность выбора открываемого документа при помощи диалога, который реализуется в виде, например, метода **DialogOpen**. Сохранение данных в этом случае может выполняться методом **XXXXSave** объекта **Document** или класса приложения.

Для выполнения операций открытия и сохранения документа в файл данных используются стандартные диалоги операционной системы, реализуемые с помощью системных функций **GetOpenFileName** и **GetSaveFileName**. В приложении В приведен примерный вид классов **DialogOpen** и **DialogSave**, с помощью которых можно управлять стандартным диалогом.

Для управления стандартными диалогами приложение должно определять путь к программе, который указывает на расположение файлов данных по умолчанию. Для этого класс приложения определяет закрытую переменную **pAppPath**, которая при старте приложения получает значение каталога программы (конструктор класса **Application**):

```
pAppPath = App.Path & "\"
```

Для того, чтобы другие классы могли воспользоваться этой переменной, класс приложения объявляет **Friend** свойство для чтения:

```
Friend Property Get AppPath() As String
    AppPath = pAppPath
End Property
```

Если приложение сохраняет свои данные в базе данных, класс приложения объявляет также **Friend** свойство, возвращающее ссылку на класс базы данных, например, так:

```
Friend Property Get DB() As Database
    Set DB = pDatabase
End Property
```

Здесь **pDatabase** — это закрытая переменная класса **Application**. Эта переменная инициализируется в конструкторе класса Application, при этом используется переменная **pAppPath**, указывающая на расположение базы данных (если база данных локальная), и константа, определяющая название файла базы данных по умолчанию, например:

```
    ' Путь к программе
    pAppPath = App.Path & "\"
    ' Путь к базе данных
    pDBPath = pAppPath & csDefaultBDName
    Set pDatabase = New Database
    ' Подключение к базе данных
    pDatabase.Connect pDBPath
```

Здесь **Database** — класс, взаимодействующий с базой данных, а **pDBPath** — закрытая переменная класса **Application**.

Инициализация коллекций

Если коллекция описывает элементы, связанные с записями в базе данных, возникает проблема инициализации коллекции. Она заключается в следующем.

Метод **Add** предназначен для добавления нового элемента коллекции. При этом, поскольку добавляемый элемент должен быть связан с новой записью базы данных, метод **Add** создает эту новую запись. Следовательно, этот метод нельзя использовать для добавления элементов в коллекцию при ее инициализации, когда она должна быть пополнена элементами, которые уже связаны с записями.

Для решения этой проблемы достаточно определить **Friend** метод для добавления элемента в коллекцию, без добавления при этом новой записи в базу данных. На самом деле лучше выполнить начальную инициализа-

цию коллекции при помощи специального **Friend** метода **InitFromDB**. Его суть заключается в том, чтобы сначала прочитать список всех идентификаторов записей в базе данных, а затем для каждого идентификатора создать новый элемент коллекции и включить его в базовую коллекцию **pColl** непосредственно. При этом получается следующий стандартный вид описываемого метода:

```
Friend Sub InitFromDB(ByVal pDB As БАЗА_ДАННЫХ)
    Dim V As Variant, I As Long, N As Long, ID As Long, Q As ТИП
    On Error Goto GeneralError
    V = pDB.ТИПsGet
    N = UBound(V)
    For I = 1 To N
        Set Q = New ТИП
        Set Q.Application = pApplication
        Set Q.Parent = pParent
        Q.ID = V(I)
        pColl.Add Q, CStr(Q.ID)
    Next
GeneralError:
End Sub
```

При этом одновременно происходит установка объектных ссылок. Заметим, что метод **БАЗА_ДАННЫХ::ТИПsGet** должен возвращать массив в переменной типа **Variant**, причем, если массив не содержит ни одного действительного элемента, на самом деле он содержит один недействительный элемент с индексом 0. Дополнительно об этом см. «Связь объектной модели с базой данных».

При использовании формальных классов инициализация их коллекций производится обычно в конструкторах родительских объектов.

В качестве примера рассмотрим процесс инициализации формальных классов в приложении «Занятия» (см. предыдущий раздел). В этом приложении классы **Week**, **Day** и **Time** являются формальными в той или иной степени. Класс **Weeks**, описывающий коллекцию учебных недель, формирует иерархию классов **Weeks** — **Week** — **Days** — **Day** — **Times** — **Time**, однозначно определяющих возможные времена занятий. Количество генерируемых этим классом объектов зависит от количества учебных недель, которое, в свою очередь, задается либо датами первого и последнего дня учебного периода, либо числом, — количеством учебных недель, — которое рассчитывается классом **Application**.

Инициализация коллекции **Weeks** выполняется в этом приложении специальным методом коллекции **Init**. Если класс **Weeks** не определяет

свойства, задающие начало и конец учебного периода, то тогда количество учебных недель может быть задан как параметр метода **Init**.

В следующем примере показан вариант создания коллекция **Weeks** в конструкторе класса **Application**:

```
Set pWeeks = New Weeks
With pWeeks
    Set .Application = Me
    Set .Parent = Me
    .DateStart = pDateStart
    .DateEnd = pDateEnd
    .Init
End With
```

Здесь закрытые переменные **pDateStart** и **pDateEnd** класса приложения определяют учебный период, а коллекция **Weeks** определяет соответствующие свойства. Если даты начала и конца учебного периода совпадают, то коллекция **Weeks** не создает никакой иерархии до момента определения этих параметров и повторного вызова метода **Init** средствами, например, пользовательского интерфейса. Если даты начала и конца учебного периода определены, коллекция **Weeks** выстраивает указанную выше иерархию классов, начиная с создания необходимого количества учебных недель (метод **Init** коллекции **Weeks**):

```
Dim I as Long, Q As Week
For I = 1 To pWeeksCount
    Set Q = New Week
    Set Q.Application = pApplication
    Set Q.Parent = pApplication
    pColl.Add Q, CStr(I)
Next
```

Здесь **pWeeksCount** — число учебных недель. При создании нового объекта **Week** срабатывает конструктор, который создает коллекцию **Days**:

```
Set pDays = New Days
```

Конструктор класса **Days** формирует элементы коллекции. Количество элементов коллекции задается константой 7 (число дней в неделе):

```
Dim I as Long, Q As Day
For I = 1 To 7
    Set Q = New Day
    pColl.Add Q, CStr(I)
Next
```

При создании нового объекта **Day** срабатывает конструктор, который создает коллекцию **Times**:

```
Set pTimes = New Times
```

Конструктор класса **Times** формирует элементы коллекции. Количество элементов коллекции здесь также может быть задано константой:

```
Dim I as Long, Q As Time
For I = 1 To 8
    Set Q = New Time
    pColl.Add Q, CStr(I)
Next
```

Поскольку в конструкторе класса объектные переменные класса **pApplication** и **pParent** еще не установлены, установка объектных ссылок множества созданных объектов выполняется при установке свойства **Application** объекта **Weeks** в конструкторе класса **Application**. При этом процедура установки свойства **Application** объекта **Weeks** модифицируется примерно следующим образом:

```
Friend Property Set Application(ByVal NewValue As Application)
    Set pApplication = NewValue
    Dim Q As Week
    For Each Q In Me
        Set Q.Application = pApplication
    Next
End Property
```

В свою очередь, процедура установки свойства **Application** класса **Week** вызывает установку свойства **Application** объекта **Days** и так далее до установки этого свойства во всех созданных объектах. Свойство **Parent** может быть установлено либо аналогичным способом, либо одновременно с установкой свойства **Application**.

Заметим, что в конце построения иерархии находятся объекты класса **Lesson**, множество которых инициализируется при считывании данных из файла хранения данных. При этом для создания объектов класс **Lessons** может определять метод **InitFormDB**, аналогичный описанному выше. Поскольку в объектной модели есть две коллекции **Lessons**, возникает вопрос — как инициализировать вторую коллекцию (ту, которая находится на уровне класса приложения)? Решением может быть добавление в эту коллекцию объектов в методах **Add** коллекций **Lessons** классов **Time** с формированием сложного составного ключа.

Взаимные ссылки между объектами

Между всеми объектами объектной модели должны быть установлены взаимные ссылки **Application** и **Parent**. Это позволяет объектам использовать свойства и методы класса **Application** и класса родительского объекта.

Свойство **Application** определяется как **Public** на чтение и **Friend** на установку. Это означает, что установку свойства берет на себя сервер приложения (классы приложения), а пользователь объектной модели может только читать это свойство, чтобы получить доступ к классу приложения из любого другого класса. Пример оформления свойства **Application**:

```
' Ссылка на класс приложения
Private pApplication As Application

Public Property Get Application() As Application
    Set Application = pApplication
End Property

Friend Property Set Application(ByVal NewValue As Application)
    Set pApplication = NewValue
End Property
```

Свойство **Parent** оформляется аналогичным образом, однако тип этого свойства должен быть **Object** для того, чтобы объекты могли быть прикреплены к разным родительским объектам. В отдельных случаях, когда родительским объектом может быть только однозначно определенный объект, внутренняя переменная **pParent**, которая хранит значение свойства, может иметь тип класса этого объекта. Обычный способ определения этого свойства приведен ниже.

```
' Ссылка на родительский класс
Private pParent As Object

Public Property Get Parent() As Object
    Set Parent = pParent
End Property

Friend Property Set Parent(ByVal NewValue As Object)
    Set pParent = NewValue
End Property
```

Со свойством **Parent** связана проблема установления отношений между классами в объектной модели, содержащей коллекции. По своей сути свойство **Parent** определяет объект, стоящий в иерархии на ступень выше. Однако, если объект является элементом коллекции, коллекция не является

при этом родительским объектом. Вместо этого в качестве родительского объекта выступает класс, содержащий коллекцию.

Возьмем в качестве примера приложение «Рецепты», рассмотренное выше. Объект класса **Component** должен принадлежать не своей коллекции, а объекту класса **Recipe** — потому что он является составной частью рецепта. Иначе говоря, *коллекции являются формальными классами*, необходимыми лишь для организации других классов и обеспечения правильного функционирования объектной модели.

В этом случае, если, например, объекту класса **Component** необходимо получить доступ к методу или свойству своей коллекции, он обращается к ней через свойство **Parent**, например, так:

```
pParent.Components.Метод
```

Чтение Friend свойств родительских объектов

Использование свойства **Parent** может вызвать некоторые затруднения внутри самой объектной модели (внутри сервера). Дело в том, что объектная модель определяется совокупностью открытых (**Public**) свойств и методов классов. **Friend** свойства и методы в ней отсутствуют.

Предположим, что объекту класса **Lesson** необходимо получить идентификаторы всех вышестоящих классов в объектной модели приложения «Занятия», приведенной на рисунке 13. Поскольку идентификаторы объектов являются внутренней особенностью классов, невидимой пользователю и отсутствующей в объектной модели, прямое обращение к классам объектной модели через переменную **pParent**, имеющую тип **Object**, невозможно, поскольку в этом случае используется интерфейс диспетчеризации. Например, в рассматриваемой объектной модели родительским объектом класса **Lesson** является класс **Time**. Предположим, что этот класс определяет **Friend** свойство **ID**, являющееся идентификатором объекта, следующим образом (часть свойства, устанавливающая значение внутренней переменной **pID**, может отсутствовать):

```
Friend Property Get ID() As Long
    Set ID = pID
End Property
```

Тогда следующий код, в котором предпринимается попытка извлечь идентификатор родительского объекта, является недопустимым:

```
Dim pParentID As Long
pParentID = pParent.ID
```

В данном простом случае для решения проблемы достаточно использовать временную переменную типа класса родительского объекта, которую необходимо инициализировать ссылкой на родительский объект **pParent**, и использовать ее, поскольку в этом случае происходит связывание через **vtable**:

```
Dim P As Time, pParentID As Long
Set P = pParent
pParentID = P.ID
```

Аналогично можно получить свойство **ID** родительского объекта для объекта **Time** (внутри объекта **Time**). Чтобы получить значение этого свойства внутри объекта **Lesson**, удобно было бы использовать конструкцию типа:

```
pParentParentID = pParent.Parent.ID
```

однако в силу указанных выше причин она опять недопустима. Для решения этой проблемы можно, например, использовать специальное **Friend** свойство **ParentID** объекта **Time**, например:

```
Friend Property Get ParentID() As Long
    Dim P As Day
    Set P = pParent
    ParentID = P.ID
End Property
```

Аналогичным образом могут быть получены **Friend** свойства и вышестоящих в иерархии объектов.

Функции приложения

Функции приложения — это набор выполняемых им действий. Автоматизированное приложение предоставляет свои функции через объектную модель и через пользовательский интерфейс, который состоит из основного меню приложения, контекстных меню и элементов управления. Создание пользовательского интерфейса приложения подробнее описывается в разделе «Основы создания интерфейса приложения».

В автоматизированном приложении все действия по управлению приложением осуществляются через его объектную модель — для этой цели она создается. При этом зачастую возникают стандартные проблемы связывания действий пользователя через интерфейс приложения с создаваемой объектной моделью.

Интерфейс приложения Windows функционирует на основе сообщений, посылаемых тому или иному окну приложения. Обработка этих сообщений осуществляется на языке *MSVB* в так называемых процедурах обработки событий. Основная задача при проектировании автоматизированного приложения — связать процедуры обработки событий элементов интерфейса с вызовами свойств и методов объектной модели.

В качестве примера рассмотрим упоминавшееся ранее приложение **`RECIPEE`**. Функциями данного приложения являются:

1) Функции по управлению базой данных:

- создание новой базы данных;
- открытие указанной базы данных;
- сохранение копии базы данных.

2) Функции по управлению справочниками:

- добавление, изменение, удаление ингредиента;
- добавление, изменение, удаление единицы измерения;

3) Функции по управлению рецептами:

- добавление, изменение, удаление рецепта;
- добавление, изменение, удаление компонента.

4) Функции по управлению приложением:

- завершение работы приложения;
- управление окном приложения.

Рассматривая перечисленные группы функций, нужно отнести их к тем или иным классам приложения и связанными с ними элементами интерфейса.

Функции по управлению базой данных

Функции по управлению базой данных относятся к приложению в целом, поэтому они реализуются классом приложения и главным окном приложения. Для этой цели в класс приложения должны быть добавлены методы, например **`FileNew`**, **`FileOpen`**, **`FileSaveAs`**, а в главном окне приложения должно быть сформировано, например, меню «Файл» с подпунктами «Новый», «Открыть…», «Сохранить как…» и «Закрыть». При этом подпункт меню «Закрыть» относится к группе функций по управлению приложением, однако по сложившимся представлениям по управлению приложением он относится к меню «Файл».

Связь действий пользователя с объектной моделью приложения в этом случае производится вызовом методом класса приложения в событиях выбора пунктов меню главного окна.

В следующем фрагменте показан примерный код события щелчок меню «Файл — Новый»:

```
Private Sub mnuFileNew_Click
    FileNew
End Sub
```

Заметим, что при сохранении данных в файл управление базой данных заменяется управлением файлами данных, однако функции приложения при этом остаются.

Функции по управлению справочниками

Функции по управлению справочниками выполняются методами коллекций справочников. С каждой коллекцией в этом случае связывается диалоговое окно для управления справочником, вызываемое методом **Show** коллекции. В соответствии с функциями приложения диалоговое окно справочника должно иметь кнопки для добавления, изменения и удаления элемента коллекции. При этом перечень всех элементов коллекции отображается в списке. Выбранный элемент коллекции отображается в поле над списком (рисунок 14).

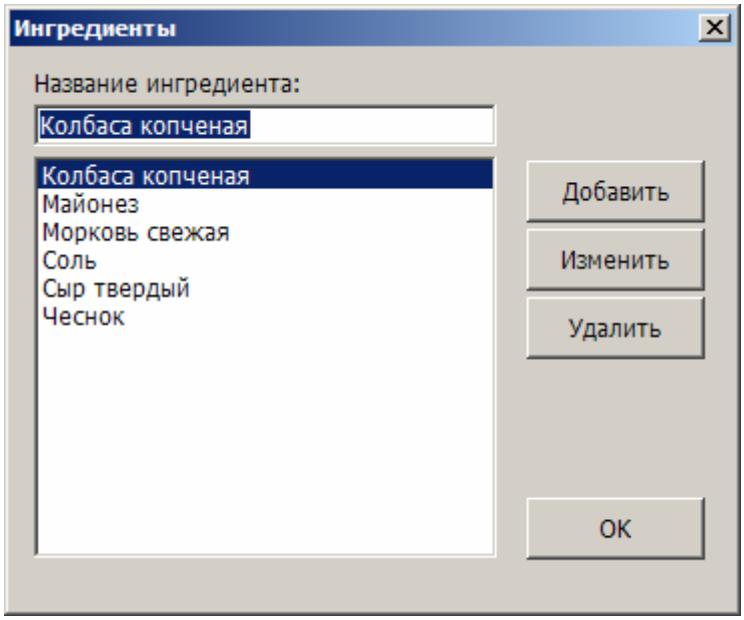

Рисунок 14 — Общий вид диалогового окна коллекции

Действия выполняются над выбранным элементом. Его можно удалить и изменить. При нажатии кнопки «Добавить» в коллекцию добавляется элемент, записанный в поле. После выполнения любого действия кол-

лекция, возможно, перечитывается из базы данных, а список элементов перечитывается из коллекции.

Список элементов может быть упорядоченным по алфавиту. Для этого у списка устанавливается свойство **Sorted**. При этом возникает проблема установления соответствия элемента списка элементу коллекции. Она решается при помощи свойства списка **ItemData**, элементы которого принимают значения идентификаторов записей в базе данных, являющихся ключами коллекции. При добавлении элемента в список одновременно в свойство **ItemData** записывается идентификатор. В следующем фрагменте приведен примерный код добавления элемента в список:

```
With List1
    .AddItem Название_элемента
    .ItemData(.NewIndex) = Идентификатор_элемента
End With
```

Для заполнения списка элементами коллекции в классе окна должен быть предусмотрен метод **DisplayItems**, упоминавшийся ранее.

Поле выбранного элемента обновляется каждый раз, когда пользователь щелкает в список, примерно следующим образом:

```
With List1
    If (.ListIndex < 0) then Exit Sub
    Text1.Text = .List(.ListIndex)
End With
```

Здесь предполагается, что название поля **Text1**.

Для выполнения действий с коллекцией, таких, как добавление, изменение и удаление элементов, окно коллекции должно иметь на нее ссылку. Эта ссылка устанавливается в методе **Show** коллекции (пример см. раздел «Метод Show»).

Функции по управлению рецептами

Управление рецептами в значительной степени зависит от модели приложения *Windows*. При использовании многодокументного интерфейса каждый отдельный рецепт располагается в своем окне. При определенных условиях это может потребовать внести изменения в объектную модель приложения. Если, например, коллекция рецептов сохраняется в базе данных, то тогда коллекция **Recipes** объектной модели на рисунке 12 не может отражать открытые для просмотра рецепты, и не может быть сокращена до количества открытых для просмотра рецептов.

В этом случае в объектную модель добавляется класс открытого документа, связанного с определенным открытым дочерним окном, и соответствующая коллекция документов.

Если для построения приложения используется диалоговая модель или одно-документный интерфейс, вводить дополнительные классы нет необходимости, поскольку в приложении нет соответствующих сущностей. В любом случае, однако, возникает проблема выбора для отображения конкретного рецепта.

Для выбора рецепта либо главное окно приложения содержит список всех рецептов, либо используется диалоговое окно, которое содержит этот список. Управление рецептами в первом случае выполняется при помощи меню приложения, а во втором — кнопками диалогового окна.

Сохранение данных

Данные приложения могут быть сохранены в локальной или удаленной базе данных, в файле специального вида, в виде структурированного хранилища, в виде записей в реестре *Windows*.

Сохранение данных в виде структурированного хранилища является сложной задачей, выходящей за рамки учебного курса, и поэтому в настоящем пособии не рассматривается.

Сохранение данных в реестре Windows

Сохранение данных приложения в реестре операционной системы предпочтительнее, если количество сохраняемых элементов исчисляется единицами. Для сохранения и чтения данных в языке *MSVB* можно использовать специальные функции **`SaveSetting`** и **`GetSetting`**. Для идентификации элемента данных используется три строковых параметра, называемые приложением (*AppName*), разделом (*Section*) и ключом (*Key*). Сохраняемые значения могут иметь произвольный тип.

Пример сохранения строкового параметра:

```
SaveSetting csAppName, csMain, csInitDir, pInitDir
```

Здесь для задания приложения, секции и ключа используются строковые константы **`csAppName`**, **`csMain`**, и **`csInitDir`**, а сохраняется закрытая переменная класса **`pInitDir`**, описывающая начальный каталог для стандартного диалога.

Пример чтения того же параметра:

```
pInitDir = GetSetting(csAppName, csMain, csInitDir, pAppName)
```

Последний параметр функции **GetSetting** — значение, возвращаемое функцией по умолчанию (при отсутствии записи в реестре). Это параметр не является обязательным. При отсутствии как читаемого параметра так и значения по умолчанию возвращается пустая строка.

Сохранение данных в файл специального вида

Под файлом специального вида здесь понимается любой файл, предназначенный для сохранения данных приложения в двоичном или текстовом формате. Фактически данный файл при этом становится документом приложения в терминах операционной системы, и требует регистрации данного типа файла в реестре операционной системы *Windows*. Одновременно приложение конструируется таким образом, чтобы файл данных открывался так же, как и приложение (при помощи, например, двойного щелчка на значок файла, или при помощи меню «Открыть»). Дополнительно с данным типом файла обычно сопоставляется значок документа.

Для описания данных в текстовом формате удобно использовать язык разметки документа, например XML. Теги XML описывают те или иные элементы данных, которые могут иметь иерархическую структуру. В следующем примере описывается документ, состоящий из объекта типа «Точка» с двумя координатами, равными в примере значениям 2 и 3:

```
<document>
  <body>
    <point>
      <X>2</X>
      <Y>3</Y>
    </point>
  </body>
</document>
```

Каждому тегу, описывающему элемент данных, здесь соответствует закрывающий тег, обозначаемый с помощью знака «слеш». Закрывающий тег заканчивает описание данного элемента данных.

Текстовые файлы данных отличаются тем, что их можно редактировать независимо от приложения при помощи обычных текстовых редакторов типа Блокнот. Двоичные файлы более закрыты в этом отношении, чем текстовые, так как изменить их, не нарушая целостность данных, можно только с помощью образующего их приложения. Открытость текстового файла может оказаться неприемлемым способом сохранения данных, когда данные должны быть защищены от изменения (когда необходимо обеспечить их целостность или конфиденциальность). В некоторых случаях при

этом можно использовать криптографические средства, а если необходимо просто предотвратить «ручное» исправление файла данных, достаточно записать файл данных в двоичном виде.

Текстовый файл при обновлении (изменении) переписывается целиком заново. Двоичный файл может иметь структуру, которая позволяет переписывать отдельные части файла. Иногда следует выработать политику сохранения старых версий файла данных. Она заключается в том, что при каждой сессии работы с приложением предыдущий файл данных сохраняется в виде так называемой *backup* копии, а результат работы текущей сессии формирует полностью новый файл данных.

При разработке файла данных следует учитывать, является ли набор данных приложения фиксированным по объему, или меняется в ходе работы с приложением. Если объем сохраняемых данных постоянен, внутренняя структура файла данных может быть жестко задана. Если же объем данных меняется от одной сессии работы с приложением к другой, следует продумать способ добавления новых данных. В некоторых случаях при этом возможна организация файла данных в виде совокупности одинаковых по формату записей.

При разработке файла данных следует также продумать, какие данные сохраняются. Если модель приложения диалоговая, то скорее всего сохраняются все данные приложения. Если модель приложения одно или многодокументная, то данные документа сохраняются обычно в отдельный файл данных. В этом случае приложение имеет меню «Файл», предназначенное для управления файлами данных и связанными с ними объектами. Дополнительно см. «Функции приложения».

Сохранение данных в базе данных

Базы данных предназначены для упорядоченного хранения данных, и сохранение данных приложения в базе данных представляется естественным решением во многих практических случаях.

Для работы с базами данных есть множество практических приемов и методов, использующих разнообразные средства. В настоящем документе описывается работа с локальной базой данных Microsoft Access версии не ниже 2000 при помощи интерфейса *ADO* версии не ниже 2.1.

Предполагается, что интерфейс приложения не использует специальные элементы управления для связи с таблицами или полями в базе данных, так как это ведет к необходимости разработки программы установки приложения.

Связь объектной модели с базой данных

Если данные приложения сохраняются в базе данных, требуется установить соответствие между объектной моделью и базой данных.

Базы данных реляционного типа состоят из таблиц, которые, в свою очередь, состоят из строк (записей). Записи состоят из полей, описывающих элементы данных. Обычной моделью соответствия между базой данных и объектной моделью является следующая:

- каждой таблице в базе данных соответствует коллекция объектной модели;

- каждой записи таблицы в базе данных соответствует отдельный класс соответствующей коллекции;

- каждому полю в строке таблицы в базе данных соответствует свойство класса, описывающего запись.

Это общее правило, из которого могут быть исключения. Например, если приложение работает с несвязанными параметрами, в базе данных им может соответствовать таблица параметров **`Param`** с двумя полями — идентификатором параметра и значением параметра. При этом никакой коллекции параметров создавать в большинстве случаев нет необходимости. Примером несвязанных параметров могут служить начало и конец учебного периода в упоминавшемся ранее приложении «Занятия».

В соответствии с данными рекомендациями схема отношений в базе данных «Рецепты», соответствующая объектной модели приложения, приведенной на рисунке 12, может иметь вид, приведенный на рисунке 15.

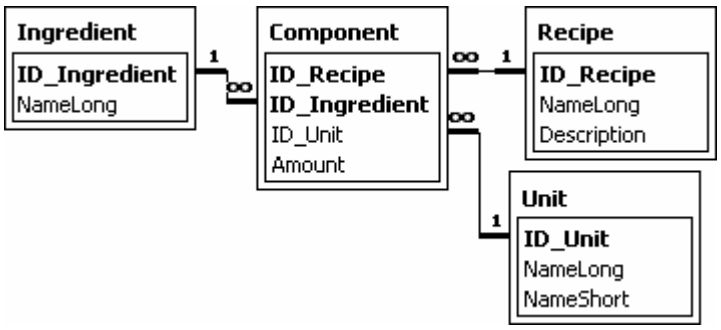

Рисунок 15 — Схема отношений базы данных «Рецепты»

Возникает вопрос, как установить соответствие между значениями элементов данных в базе данных и значениями свойств соответствующих классов. Для решения этого вопроса в каждый класс, представляющий собой строку в таблице базы данных, добавляется **`Friend`** свойство **`ID`**, которое соответствует уникальному идентификатору записи в базе данных (на-

пример, ключевому полю типа счетчик). При инициализации класса устанавливается значение только этого свойства. При обращении к другим свойствам класса выполняется запрос к базе данных с целью получить значение из базы данных или наоборот — изменить его.

Например, для чтения и записи свойства **Description** класса **Recipe** может быть использован следующий примерный код:

```
' Чтение свойства из базы данных
Public Property Get Description() As String
    Description = pApplication.DB.RecipeDescriptionGet(pID)
End Property
' Запись свойства в базу данных
Public Property Let Description(ByVal NewValue As String)
    pApplication.DB.RecipeDescriptionSet pID, NewValue
End Property
```

Для понимания данного фрагмента кода дополнительно нужно иметь представление о методах класса базы данных.

Класс базы данных

Для установления соединения с базой данных и выполнения запросов к ней в сервер добавляется закрытый (**Private**) класс **Database**.

Класс содержит две закрытых переменных уровня модуля: **conn** для хранения соединения с базой данных и **pConnected** для хранения признака установления соединения:

```
Private conn As ADODB.Connection
Private pConnected As Boolean
```

Метод **Connect** выполняет соединение с базой данных:

```
Public Function Connect(ByVal Path As String) As Boolean
    Disconnect
    On Error GoTo OpenError
    Set conn = New ADODB.Connection
    With conn
        .Provider = "Microsoft.Jet.OLEDB.4.0"
        .ConnectionString = Path
        .Open
    End With
    Connect = True
    pConnected = True
    Exit Function
OpenError:
End Function
```

Здесь параметр **Path** — это путь к базе данных. Дополнительно см. раздел «Класс Application».

Метод **Disconnect** разрывает соединение с базой данных:

```
Public Sub Disconnect()
    If conn Is Nothing Then Exit Sub
    If (conn.State = adStateOpen) Then
        conn.Close
    End If
    Set conn = Nothing
    pConnected = False
End Sub
```

Свойство **Connected** возвращает признак установления соединения:

```
Public Property Get Connected() As Boolean
    Connected = pConnected
End Property
```

Другие методы класса **Database** предназначены для добавления и удаления записей и извлечения и установки полей в таблицах базы данных. В качестве примера рассмотрим свойство **Description** класса **Recipe** и соответствующие ему методы класса **Database**. Следующий метод извлекает значение поля **Description** из таблицы **Recipe**:

```
Public Function RecipeDescriptionGet(ByVal IDRecipe As Long) As
String
    Dim RS As New ADODB.Recordset, Q As String, V As Variant
    On Error GoTo GeneralError
    Q = "SELECT * FROM " & csRecipe & " WHERE " & _
        csIDRecipe & "=" & CStr(IDRecipe)
    With RS
        .Open Q, conn, adOpenKeyset, adLockOptimistic, adCmdText
        V = .Fields(csDescription)
        If Not IsNull(V) Then
            RecipeDescriptionGet = CStr(V)
        End If
        .Close
    End With
    Exit Function
GeneralError:
End Function
```

Метод принимает параметр **IDReicpe** — идентификатор записи рецепта в таблице, и возвращает значение поля **Description**. Метод формирует запрос на извлечение записи с данным идентификатором, открывает новый

объект **Recordset** и читает поле **Description** в переменную типа **Variant**. Это необходимо для того, чтобы не возникло ошибки при чтении значения типа **Null**. Далее в методе проверяется, является ли считанное значение **Null**, и если нет, то считанное значение возвращается.

Обратим внимание на способ формирования имени метода. Сначала используется название таблицы, затем название поля, затем глагол, указывающий действие. Следующий метод записывает новое значение данного поля в базу данных:

```
Public Function RecipeDescriptionSet(ByVal IDRecipe As Long, ByVal
NewValue As String) As Boolean
    Dim RS As New ADODB.Recordset, Q As String
    On Error GoTo GeneralError
    Q = "SELECT * FROM " & csRecipe & " WHERE " & _
        csIDRecipe & "=" & CStr(IDRecipe)
    With RS
        .Open Q, conn, adOpenKeyset, adLockOptimistic, adCmdText
        .Fields(csDescription) = NewValue
        .Update
        .Close
    End With
    RecipeDescriptionSet = True
    Exit Function
GeneralError:
End Function
```

Отличия заключаются в том, что поле не читается, а записывается, а для внесения изменений в базу данных вызывается метод **Update**. В приведенных примерах используются строковые константы для названий полей.

Следующий пример показывает, как добавляется новая запись о рецепте. Первая часть метода **RecipeAdd** проверяет наличие записи с указанным названием **NameLong**:

```
Public Function RecipeAdd(ByVal NameLong As String) As Long
    Dim RS As New ADODB.Recordset, Q As String, E As Long
    On Error GoTo GeneralError
    Q = "SELECT COUNT(*) AS Z FROM " & csRecipe & " WHERE " & _
        csNameLong & "='" & NameLong & "'"
    With RS
        .Open Q, conn, adOpenKeyset, adLockOptimistic, adCmdText
        E = .Fields("Z")
        .Close
        If (E <> 0) Then Exit Function
```

Если запись существует, дальнейшие действия не выполняются. Заметим, что поле **NameLong** в базе данных является индексированным с запретом совпадений. Если запись не существует, добавляется новая запись с одним заполненным параметром **NameLong**:

```
        .Open csRecipe, conn, adOpenKeyset, adLockOptimistic,
adCmdTable
        .AddNew
        .Fields(csNameLong) = NameLong
        .Fields(csDescription) = ""
        .Update
        RecipeAdd = .Fields(csIDRecipe)
        .Close
    End With
    Exit Function
GeneralError:
End Function
```

Здесь **csRecipe** — строковая константа, описывающая название таблицы **Recipe**. Возвращаемое значение метода — идентификатор добавленной записи. Заметим, что метод может также возвращать идентификатор существующей записи, если запись с указанным значением параметра **NameLong** существует.

Следующий пример показывает, как удаляется запись о рецепте:

```
Public Function RecipeRemove(ByVal IDRecipe As Long) As Boolean
    Dim CMD As New ADODB.Command, Q As String
    On Error GoTo GeneralError
    Q = "DELETE * FROM " & csRecipe & " WHERE " & _
        csIDRecipe & "=" & CStr(IDRecipe)
    With CMD
        Set .ActiveConnection = conn
        .CommandText = Q
        .Execute
    End With
    RecipeRemove = True
    Exit Function
GeneralError:
End Function
```

Здесь вместо объекта **Recordset** используется объект **Command**, более подходящий для данного действия.

Методы для добавления и удаления записей, а также для извлечения и модификации полей других таблиц принципиально ничем не отличаются от приведенных выше.

Чтение коллекций

При инициализации приложения коллекции объектной модели должны быть заполнены строками из соответствующих таблиц базы данных. Для этой цели коллекции должны иметь **Friend** методы **InitFromDB**, внутри которых из таблиц базы данных читаются все идентификаторы строк, создается необходимое количество классов и устанавливается их свойство **ID**. Пример метода **InitFromDB** приведен в разделе «Инициализация коллекций». Здесь приводится пример метода класса базы данных, который формирует массив идентификаторов записей таблицы **Recipe**:

```
Public Function RecipesGet() As Variant
    Dim RS As New ADODB.Recordset, V As Variant, I As Long, N As
Long
    On Error GoTo GeneralError
    ReDim V(0) As Long
    With RS
        .Open csRecipe, conn, adOpenKeyset, adLockOptimistic,
adCmdTable
        .MoveFirst
        N = .RecordCount
        ReDim V(N)
        For I = 1 To N
            V(I) = CLng(.Fields(csIDRecipe))
            .MoveNext
        Next
        .Close
    End With
    RecipesGet = V
    Exit Function
GeneralError:
End Function
```

Значения идентификаторов возвращаются в массиве **Variant**, который предварительно устанавливается в нулевую размерность. При ошибке, а также в случае отсутствия записей в таблице возвращается массив с нулевой размерностью.

Создание базы данных

Для программного создания базы данных используется интерфейс *ADOX* (ссылка *Microsoft ADO Ext. 2.X for DLL and security*). В классе базы данных для этой цели может быть определен метод **CreateNew**. Метод сначала создает саму базу данных (объект **Catalog**):

```
    Dim C As New Catalog, T As Table, X As Index, K As Key
    On Error GoTo GeneralError
    C.Create "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & Path
```

Здесь **Path** — параметр метода, указывающий на путь к базе данных. Далее создаются таблицы, поля, индексы и ключи. В качестве примера рассмотрим создание таблицы **Units** (единицы измерения). Сначала создаются поля таблицы, в том числе ключевое поле типа «Счетчик»:

```
Set T = New ADOX.Table
With T
    .Name = csUnit
    Set .ParentCatalog = C
    .Columns.Append csIDUnit, adInteger
    .Columns(csIDUnit).Properties("AutoIncrement") = True
    .Columns.Append csNameLong, adVarWChar, 50
    .Columns.Append csNameShort, adVarWChar, 10
End With
```

Далее создаются индексы, в примере — ключевое поле, а также индексы для поиска по короткому и длинному названиям:

```
Set X = New ADOX.Index
With X
    .Name = csUnit & "_" & csIDUnit
    .PrimaryKey = True
    .Columns.Append csIDUnit
End With
T.Indexes.Append X
Set X = New ADOX.Index
With X
    .Name = csUnit & "_" & csNameLong
    .Unique = True
    .Columns.Append csNameLong
End With
T.Indexes.Append X
Set X = New ADOX.Index
With X
    .Name = csUnit & "_" & csNameShort
    .Unique = True
    .Columns.Append csNameShort
End With
T.Indexes.Append X
```

В заключение таблица добавляется в базу данных

```
C.Tables.Append T
```

Другие таблицы создаются аналогичным образом.

Следующий пример показывает создание внешнего ключа и установление ограничения целостности между таблицами **Ingredient** и **Unit**:

```
Set K = New ADOX.Key
With K
    .Name = csIngredient & csUnit
    .Type = adKeyForeign
    .RelatedTable = csUnit
    .Columns.Append csIDUnit
    .Columns(csIDUnit).RelatedColumn = csIDUnit
    .UpdateRule = adRICascade
    .DeleteRule = adRINone
End With
C.Tables(csComponent).Keys.Append K
```

В завершение метод **CreateNew** может формировать предопределенные записи в таблицах. В следующем примере добавляется несколько записей в таблицу **Units** (используется интерфейс *ADO*):

```
If Connect(Path) Then
    Dim RS As New Recordset
    With RS
        .CursorLocation = adUseClient
        .Open csUnit, conn, _
            adOpenKeyset, adLockBatchOptimistic, adCmdTable
        .AddNew
        .Fields(csNameLong) = "не установлено"
        .Fields(csNameShort) = csNAUnit
        .AddNew
        .Fields(csNameLong) = "грамм"
        .Fields(csNameShort) = "г"
        .AddNew
        .Fields(csNameLong) = "килограмм"
        .Fields(csNameShort) = "кг"
        .UpdateBatch
        .Close
    End With
End If
```

Здесь применяется пакетное обновление записей (при помощи метода **UpdateBatch**), для чего должен быть использован курсор на стороне клиента (свойство **CursorLocation** должно быть установлено в значение **adUseClient**).

Основы создания интерфейса приложения

Создание интерфейса приложения — чрезвычайно сложная и многогранная задача, для выполнения которой требуются скорее художники, композиторы, методисты, дизайнеры и психологи, нежели программисты. В настоящем пособии не ставится цель раскрыть все аспекты этой весьма важной стороны любого программного обеспечения, которое взаимодействует с пользователем. Существует, однако, несколько основополагающих правил создания интерфейса приложения, которым программист должен следовать и, следовательно, должен их знать.

Для более полного понимания принципов взаимодействия пользователя с программой (и вообще с технически устройством) рекомендуется изучение, например, фундаментального труда Джефа Раскина [4]. Достаточно подробное описание элементов стандартного интерфейса приведено в книге [5], которая, по большей части, является переводом рекомендаций фирмы *Microsoft*, приведенные в [6]. Дополнительно рекомендуется также изучить исследования *Microsoft* в области интерфейса приложения [7].

Неколичественные характеристики интерфейса

Интерфейс — это то, что видит пользователь на экране компьютера, когда «открывает» приложение. Интерфейс, с точки зрения пользователя, очевидно, и есть само приложение. Чтобы пользователь смог начать работу с программой без предварительного обучения, а также для наиболее продуктивного взаимодействия с ней во время дальнейшей эксплуатации, интерфейс программы должен удовлетворять некоторым критериям, которые являются субъективными.

Узнаваемость

Интерфейс приложения прежде всего должен обеспечивать «*узнаваемость*» навыков, полученных пользователем в результате его опыта работы с другими приложениями. Для обеспечения «*узнаваемости*» интерфейс приложения строится с соблюдением определенных общепринятых норм. Эти нормы включают в себя принципы построения основного меню приложения, принципы создания диалогов и основы использования элементов управления, а также принципы конструирования окон приложения, основанные на психологических и эргономических ограничениях человеческого восприятия информации, а также сложившиеся исторически в ходе эволюционного развития программных систем. Узнаваемость интерфейса обеспечивается *согласованностью* его элементов с операционным окружением, в котором приложение функционирует.

Понятность

Понятность, иначе называемая *интуитивной понятностью* или просто *интуитивностью* интерфейса, обозначает свойство интерфейса отображать элементы данных и элементы управления ими в соответствии с особенностями восприятия, сложившимися у пользователя в результате его определенного культурного воспитания и сложившегося мировоззрения. Понятный интерфейс позволяет пользователю интуитивно определять назначение его элементов и вероятные действия (или алгоритмы действий) программы.

На практике это свойство является одним из наиболее трудно достижимых. Это связано с особенностью проецирования элементов интерфейса во внутренний язык, которым пользуются конкретные пользователи для направления своих действий и рассуждений во время работы.

Предсказуемость

Предсказуемость интерфейса означает его определенное (адекватное) поведение в ответ на определенные действия пользователя. Например, щелчок на элемент управления типа «флажок» включает или выключает последний, а не вызывает, например, появление диалога. Часто разработчики, пытаясь, видимо, упростить интерфейс, прячут его элементы до момента выполнения пользователем определенного действия. В результате в окне приложения неожиданно для пользователя появляются новые элементы управления, при этом пользователь не в состоянии вспомнить процедуру, приведшую к их появлению, а также не может предсказать действия, которые приведут к их исчезновению.

Для обеспечения предсказуемости программисту требуется в основном использовать элементы интерфейса по их прямому назначению, не возлагать на один элемент интерфейса несколько функций (которые варьируются в зависимости от состояния программы), более тщательно анализировать процесс диалога пользователя с программой.

Простота

Простота — обобщенное понятие узнаваемости, понятности и предсказуемости интерфейса, обозначающее элементарность выполняемых при помощи интерфейса отдельных операций. Простота интерфейса обеспечивает также его привлекательность — пользователю приятно осознавать, что он может легко и непринужденно управлять любыми сложными процессами. Средством обеспечения простоты является деление сложных последовательностей действий на простые и понятные шаги.

Удобство

Удобство, или естественность, также является обобщающим понятием, так как перечисленные характеристики интерфейса в конечном счете должны вести к его удобству. Удобство можно определить как рациональное с точки зрения эргономики размещение его элементов, и простое и понятное их взаимодействие, обеспечивающие композиционную стройность.

Отзывчивость

Это свойство интерфейса должно обеспечивать пользователя наглядным подтверждением его действий, и выполнять роль обратной связи.

Некоторые элементы управления сами подтверждают выполнение действия изменением своего внешнего вида (например, флажки и переключатели). Другие элементы управления остаются неизменными после того, как пользователь воспользуется ими (например, кнопки). Независимо от этого интерфейс должен показывать пользователю, что запрошенное действие *выполнено* или *выполняется*. Недопустимо оставлять пользователя в неведении относительно того, что делает программа в тот момент, когда ее интерфейс недоступен (не реагирует на действия пользователя).

При проектировании программы следует обратить особое внимание на циклические конструкции. Если их выполнение может затянуться на продолжительное время (более секунды), необходимо переконструировать цикл таким образом, чтобы программа реагировала на события, а пользователь мог в любой момент прервать затянувшуюся операцию.

Следует также обратить внимание на исключительные ситуации, которые могут возникнуть в результате выполнения операций. Сообщения, которые программа выводит пользователю, должны быть понятными, а программа должна предлагать варианты дальнейших действий.

Сюда же можно отнести способность интерфейса «прощать» ошибки пользователя. Если ошибочное действие можно отменить, то необходимо предоставить пользователю такую возможность. Если действие не может быть отменено, то тогда оно не должно вести к катастрофическим, с точки зрения пользователя, последствиям.

Эстетическая привлекательность

Интерфейс должен, что называется, «радовать глаз». Эстетическую привлекательность интерфейса обеспечивают композиция, цветовая гамма и элементы художественного оформления. Для построения хорошей композиции следует знать законы ее построения. Цветовая гамма может оказать значительное влияние на продуктивность работы пользователя, а ис-

пользование в интерфейсе цветных объектов могут улучшить его *наглядность*. Элементами художественного оформления являются линии, рамки, картинки и надписи, а также звуковое сопровождение, мультипликация и другие мультимедийные возможности программы. Эстетическую привлекательность могут обеспечивать также трехмерный вид элементов интерфейса, нестандартные элементы управления, образы объектов предметной области.

Проектирование окон

Окна приложения подразделяют на *первичные* и *вторичные*. Первичные окна являются основными окнами приложения. К ним относятся: главное окно приложения диалогового типа или типа *SDI*, главное окно и дочерние окна приложения *MDI*. Вторичными окнами являются окна модальных и немодальных диалогов (они появляются в результате действий, выполняемых во время работы с первичными окнами).

Окно является главным элементом интерфейса приложения, — все другие элементы располагаются внутри окон. Окна могут иметь постоянный или изменяемый размер, а также одинарную и двойную рамку (управляется свойством **`BorderStyle`**), плоский или объемный вид (управляется свойством **`Appearance`**).

Основные окна

Основные окна предназначены для размещения (отображения) основной информации приложения (обычно документа), и основных или всех элементов управления ею.

Основное окно *приложения диалогового типа* имеет постоянный размер. Это связано с невозможностью или нерациональностью перераспределения (повторного размещения) фиксированного набора элементов управления при изменении размера окна. При проектировании окна необходимо определить его размер таким, чтобы окно умещалось на экране в любом случае эксплуатации программы.

Есть два подхода к определению размера окна приложения диалогового типа. В первом случае размер окна имеет постоянное значение, которое не меняется в течение всего периода эксплуатации программы. Во втором случае размер окна может несколько изменяться в зависимости от фактической разрешающей способности дисплея и его размера. При этом во время старта приложения определяются характеристики дисплея и задается соответствующий им размер окна. При этом одновременно изменяются размеры элементов управления и шрифтов надписей.

Значения постоянных размеров окна зависят от потенциального круга потребителей. Если приложение разрабатывается для широкого распространения, следует учесть, что у многих пользователей могут оказаться небольшие размеры дисплея.

Дополнительно об окнах диалоговых приложений см. также разделы «Элементы управления» и «Разработка диалогов».

Для приложений типа *SDI* и *MDI* основные окна имеют изменяемый размер. В приложении типа *MDI* основными окнами являются специальное *MDI* окно, которое является главным окном приложения и может присутствовать в проекте в единственном числе, а также окна документов (дочерние окна *MDI*). Форма проекта становится дочерним окном *MDI*, если у нее установлено свойство **`MDIChild`**.

Основное окно приложения *SDI* и дочернее окно приложения *MDI* содержат *данные документа* приложения. Это могут быть, например, схемы, изображения процессов и т.п. В зависимости от элементов данных, это может потребовать от программиста некоторой изобретательности для их отображения и (или) размещения. Как правило, стандартные элементы управление мало пригодны для этой цели, и, возможно, потребуется разработка нестандартного элемента управления или нескольких таких элементов управления.

Замечено, что использование приложений типа *MDI* вызывает определенные трудности у пользователей. Фирма *Microsoft* уже давно отошла от стандартной схемы *MDI* интерфейса в приложениях *Microsoft Office*, предпочитая схему, в которой каждый документ приложения открывается в собственном главном окне приложения. При этом сохраняются полная функциональность стандартного *MDI* интерфейса — документы переключаются, например, при помощи стандартного сочетания клавиш *Ctrl+F6*.

При проектировании окна с изменяемым размером следует предусмотреть некоторый *размер окна по умолчанию* — тот, который окно будет иметь во время запуска. Как правило, этот размер устанавливается во время разработки. Необходимо помнить о том, что размеры окна не должны превышать некоторые минимальные значения, например, 800 пикселей по ширине и 600 пикселей по высоте. В противном случае может оказаться, что на некоторых компьютерах размер окна превысит размер экрана.

Во время работы пользователь может задать произвольные размеры окна, а программист в событии **`Form_Resize`** выполняет перераспределение пространства окна между элементами, отображающими документ приложения. Если все пространство окна занимает, например, нестандартный

элемент управления, то он выполняет перераспределение отображаемой информации самостоятельно, в своей процедуре **UserControl_Resize**. Чтобы это произошло, в событии основного окна **Form_Resize** следует установить новые размеры элемента управления, например, так:

```
Private Sub Form_Resize()
    UserControl1.Move 0, 0, ScaleWidth, ScaleHeight
End Sub
```

Диалоговые окна

Диалоговые окна предназначены для управления параметрами (характеристиками, настройками) приложения, а также для отображения и управления параметрами вторичных объектов программы.

Диалоговые окна всегда имеют постоянный размер. Размер окна устанавливается во время его конструирования или программно во время загрузки окна (в процедуре события **Form_Load**). Свойство **BorderStyle** диалогового окна должно иметь значение **3 — Fixed Dialog**.

Размещение элементов управления в окне диалогового типа обычно выполняется во время разработки окна. Тем не менее, правильно разместить элементы управления можно также программно в процедуре события **Form_Resize**, которое *всегда* возникает при первом появлении окна на экране. Этот способ используется также в случае, если элементы управления создаются и размещаются во время загрузки окна (когда используются, например, массивы элементов управления).

Размер окна диалога, как правило, много меньше размера основного окна приложения. Если это не так, то следует пересмотреть набор параметров, для управления которыми диалоговое окно создается. Возможно, диалог можно поделить на две или более частей, и сформировать либо отдельные диалоги, либо диалог с вкладками.

Диалоговые окна отличаются тем, что на них размещаются кнопки «OK» и «Отмена». При помощи первой кнопки установленные значения параметров подтверждаются пользователем, а при помощи второй — отменяются, а все произведенные программой изменения данных *откатываются*. Обе кнопки приводят к исчезновению окна с экрана.

Если откат данных невозможен, кнопка отмены не имеет смысла, и в этом случае ее не следует размещать в окне. Заметим также, что системная кнопка с крестиком «Закрыть», имеющаяся по умолчанию у каждого окна, является аналогом кнопки «Отменить». Поэтому, если отмена действий невозможна, то и кнопку с крестиком следует запретить. Ей управляет

свойство формы **ControlBox**. На диалоговых окнах должны отсутствовать также системные кнопки «Свернуть» и «Развернуть».

Если диалоговое окно управляет вторичным объектом (например, коллекцией объектов), кнопка «Отмена» может также не иметь смысла. Иногда в этом случае вместо кнопки «OK» размещают кнопку «Закрыть». Этот вариант не является удачным, — пользователь может предположить, что данная кнопка закрывает приложение.

У кнопки, которая подтверждает изменения параметров, должно быть установлено свойство **Default**, означающее, что нажать эту кнопку и завершить диалог можно будет также при помощи клавиши «Enter».

У кнопки, которая отменяет изменения параметров, должно быть установлено свойство **Cancel**, означающее, что нажать эту кнопку и завершить диалог можно будет также при помощи клавиши «Escape».

Если диалоговое окно используется для задания параметров, возникает проблема, когда и как параметры передать в диалог и обратно в программу. Один из возможных вариантов заключается в том, чтобы не выгружать окно из памяти, когда пользователь щелкает кнопку «OK». В этом случае последовательность действий при работе с диалоговым окном может быть такой, которая приведена в следующем примере кода:

```
' Создаем и загружаем в память форму диалога
Dim Q As New Form1
Load Q
With Q
    ' Устанавливаем значения параметров
    .SetParams Param1, Param2, . . .
    ' Выводим окно на экран модально
    .Show vbModal
    ' Если диалог закрыт кнопкой OK, считываем параметры из окна
    If .Accept Then
        .GetParams Param1, Param2, . . .
    End If
End With
' Выгружаем диалог из памяти
Unload Q
```

В примере предполагается, что процедура **SetParams** устанавливает значения параметров в элементах управления окна, процедура **GetParams** наоборот, считывает значения параметров из элементов управления, а свойство **Accept** указывает на то, какой кнопкой диалог был закрыт. В событии **Click** кнопки «OK» это свойство устанавливается в значение **True**, а

в событии формы `Form_Load` — в значение `False`. Кнопки «OK» и «Отмена» не выгружают окно, а скрывают его при помощи метода формы `Hide`.

Приведенный пример является ориентировочным. На самом деле значения параметров могут устанавливаться и считываться не с помощью процедур `SetParams` и `GetParams`, а непосредственно с элементов управления формы.

Диалоговые окна могут использоваться в модальном и немодальном режимах. Модальностью управляет параметр метода `Show` формы. Пример задания модального диалога см. выше.

Модальное диалоговое окно приостанавливает работу приложения до своего закрытия. Немодальное диалоговое окно работает параллельно с другими окнами приложения. Немодальное диалоговое окно следует размещать поверх всех других окон, для чего используется системная функция `SetWindowPos` с флагом `HWND_TOPMOST`.

Немодальное диалоговое окно не используется для управления параметрами. Характерный случай его применения — поиск и замена.

Разработка диалогов

Разработка диалогов заключается в проектировании необходимой композиции элементов управления, установления правильного порядка обхода клавишей *Tab*, определении клавиш быстрого управления, обработке событий элементов управления.

Перед проектированием диалога нужно четко определить его назначение и перечень управляемых диалогом параметров. Не следует перегружать диалог обилием функций — пользователь может просто запутаться в их назначении. Желательно, чтобы диалог управлял как можно меньшим количеством параметров, однако это не главное правило. Небольшое число параметров может привести к слишком маленькому окну, которое пользователь может «не заметить». В этом случае следует дополнить содержимое окна какими-нибудь вспомогательными параметрами или элементами, отображающими, например, результат применения устанавливаемых параметров диалога.

При проектировании формы окна следует придерживаться правила «золотого сечения» — отношение сторон прямоугольника окна должно быть примерно таким, что и отношение соответствующих сторон прямоугольника стандартного экрана телевизора — 4:3. Вытянутая в ширину или в высоту форма окна должна быть оправдана исключительной необходимостью, например, обусловлена формой отображаемого объекта.

Композиция элементов управления должна быть равномерной. Это означает, что следует разместить элементы управления так, чтобы расстояния между ними были приблизительно одинаковыми и (или) пропорциональными. Нежелательно наличие пустых областей окна, равно как и областей, в которых элементы управления «скучены».

Этого не всегда удается достичь. Можно порекомендовать использовать появляющиеся свободные области для, например, размещения пояснительных надписей, картинок или иного оформления. Однако, не следует этим злоупотреблять. По крайней мере, все диалоговые окна примерно одинакового назначения должны иметь примерно одинаковый вид, то есть примерно одинаковую композицию.

Поля окна диалога должны быть также равномерными. Нормальный размер поля имеет значение примерно 8-16 пикселей. Для улучшения зрительного восприятия диалога с объемными границами можно порекомендовать правое и нижнее поле делать на 1-2 пикселя больше (это связано с переходами цвета на границах окна).

При размещении элементов управления необходимо установить одинаковые расстояния между одинаковыми элементами управления. Для выбора самого расстояния используется понятие дискреты [5]. Дискрета — аппаратно-независимая единица измерения расстояния в диалоге, зависящая от размера стандартного шрифта диалога. В горизонтальном направлении величина дискреты выбирается равной одной четверти средней ширины символа, а в вертикальном — одной восьмой средней высоты символа. Вертикальное расстояние между кнопками должно быть равным примерно 3-4 дискретам, а вертикальное расстояние между группами кнопок принимается равным 7-8 дискретам. Эти правила могут быть применены и к другим элементам управления. Горизонтальное расстояние между элементами управления может находиться в пределах 4-8 дискрет.

Следует обратить внимание на размещение элементов управления в соответствии с их назначением. Если диалог управляет группами параметров, эти группы должны быть размещены внутри рамок, образующих логические области управления. Однако, применение рамок для одного большого по размеру элемента управления, который управляет не одним параметром, а также для диалога в целом является недопустимым.

Следует помнить о том, что пользователи «*ненавидят*» диалоги. Не случайно последние версии *Microsoft Office* (2007 года) претерпели существенное изменение — в них предпринята попытка отказаться от диалогов, как средства задания текущих параметров. Все инструменты, необходимые

пользователю для выполнения текущей операции, размещаются непосредственно в основном окне в виде, например, панелей инструментов.

В связи с этим к проектированию диалогов следует относиться очень внимательно. Нужно упрощать всеми возможными для программиста средствами чтение диалога пользователем. При этом следует знать, что обычный способ чтения информации пользователем — сверху вниз и слева направо, и размещать информацию в соответствии с этим принципом.

Другой принцип размещения информации в диалоге связан с привычными пользователю бумажными формами. Если диалог отображает такую форму для ввода данных, имеет смысл сделать ее похожей на бумажный документ с точным соблюдением формата.

После размещения элементов диалогового окна следует установить правильный порядок переключения фокуса клавишей табуляции *Tab*. Для этой цели в *MSVB* каждый элемент управления, который может получать фокус, обладает свойством **`TabIndex`**. Следует установить последовательный порядок значений этого свойства, начиная от нуля, всем элементам управления, которые могут получать фокус. При этом следует придерживаться правила обхода элементов управления сверху вниз и слева направо.

Так, если диалог предлагает элементы управления в две колонки, то сначала обходятся элементы первой колонки сверху вниз, а затем элементы второй колонки также сверху вниз.

Каждому элементу управления необходимо также назначить клавишу быстрого доступа. Для большинства элементов управления эта клавиша задается при помощи знака «амперсанд» **`&`**, предшествующего символу управления. Таким образом пользователь сможет быстро вызвать основное действие элемента управления при помощи сочетания клавиши *Alt* с клавишей быстрого доступа.

При назначении клавиш быстрого доступа следует по возможности обеспечить разные символы быстрого доступа к разным элементам управления. При этом желательно, чтобы символом быстрого доступа был тот, который является первым в надписи, например, «**`&Закрыть`**».

Элементы управления

Элементы управления можно поделить на стандартные и нестандартные (разрабатываемые пользователем). Стандартные элементы управления предназначены для быстрой организации стандартного графического пользовательского интерфейса. Недостаток стандартных элементов управления — унифицированный внешний вид, приводящий к однообразию приложений. Достоинство — легкая узнаваемость интерфейса.

Далее рассмотрены основные стандартные элементы управления *MSVB* и приведено их стандартное применение.

Надпись (*Label*). Предназначена для размещения в окне статического или динамического текста. Динамическим текст используется для вывода в окно сообщений пользователю. Может быть использована также в качестве цветного прямоугольника, при этом текст надписи удаляется. В отличие от фигуры, реагирует на события мыши.

Кнопка (*CommandButton*). Предназначена для выполнения команд, например, «Добавить», «Удалить», «Вычислить» и т.п. Кнопки могут приводить к продолжительным вычислениям, что следует учитывать.

Флажок (*CheckBox*). Предназначен для управления одиночным независимым параметром, имеющим два произвольных значения, например, «Включено» и «Выключено».

Переключатели (*OptionButton*). Предназначены для управления как минимум двумя взаимоисключающими параметрами (выбора одного из них). Применяются только в составе группы таких элементов управления. Если в окне нужно разместить несколько таких групп, следует поместить каждую отдельную группу в какой-нибудь отдельный контейнер.

Поле (*TextBox*). Используется для ввода текстовой информации. Если требуется ввести не текстовую, а числовую информацию, следует определить обработку события **KeyPress**, с помощью которой можно запретить ввод определенных символов. В следующем примере такой обработки в поле можно ввести цифры, запятую, клавишу «Забой» (*Backspace*):

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
    Select Case KeyAscii
    Case 48 To 57, 8     ' Цифры и Забой
    Case 44, 46          ' Точка становится запятой
        KeyAscii = 44
    Case Else            ' Остальное заперещено
        KeyAscii = 0
    End Select
End Sub
```

Список (*ListBox*). Используется для выбора одного или нескольких параметров из их списка. Аналогичными свойствами и назначением обладает раскрывающий список (*ComboBox*). Его отличие от обычного списка заключается в том, что он является комбинацией поля и списка.

Рамка (*Frame*). Является простым контейнером, предназначенным для логического группирования других элементов управления.

Графический контейнер (*PictureBox*). Является контейнером, предназначенным в основном для размещения графической информации. Это могут быть как картинки, так и рисунки, получаемые при помощи графических методов самого элемента управления.

Картинка (*Image*). Предназначен для размещения картинок. Другое назначение — область для фиксации событий мыши в определенном месте окна.

Фигура (*Shape*). Пассивный (не обладающий событиями) элемент графического оформления в виде цветного прямоугольника (квадрата) или овала (круга). Прямоугольник (квадрат) может иметь скругленные углы.

Линия (*Line*). Пассивный элемент графического оформления. Для получения объемной линии два элемента управления располагаются рядом, одному из них назначается белый цвет, другому — темно-серый.

Другие элементы управления в *MSVB* должны быть подключены к проекту при помощи диалога *Project — Components*. Использование подключаемых элементов управления приводит к необходимости их установки на компьютер пользователя при помощи специальной программы установки, что усложняет развертывание приложения. Тем не менее, именно в подключаемых элементах управления находятся многие важные из них, например, панели инструментов и вкладки диалогов.

Шрифт

Для выполнения надписей на элементах управления желательно использовать один и тот же шрифт для всего приложения. При этом следует, во-первых, использовать гарнитуру (название шрифта), которая есть на всех компьютерах, а во-вторых — использовать гарнитуру, которая соответствует экранному шрифту. Это общее правило.

Экранный шрифт отличается тем, что образы его символов оптимизированы для отображения на дисплее, который имеет относительно низкую разрешающую способность, иначе говоря, экранный шрифт легче читается. К таким шрифтам относятся *Tahoma* (узкий) и *Verdana* (широкий). Эти шрифты есть на любом компьютере, и они являются шрифтами типа *OpenType*, то есть масштабируемыми и сглаживаемыми по технологии *Microsoft ClearType*, что имеет значение при использовании *LCD* мониторов.

Размер шрифта следует выбирать, исходя из текущей разрешающей способности дисплея и размера экрана. Обычная разрешающая способность дисплея равна 96 dpi, однако на некоторых компьютерах она может оказаться выше, например, 120 dpi. При разрешающей способности дис-

плея 96 dpi и разрешении экрана до 1280×960 пикселей минимальный размер шрифта типа *Tahoma* должен составлять не менее 8-9 пунктов. При большей разрешающей способности или при большем разрешении экрана размер шрифта следует принимать равным не менее 9-10 пунктов.

Не следует без существенной необходимости использовать курсивное начертание — оно ухудшает читабельность надписей. Полужирное (жирное) начертание допустимо для выделения важной информации, хотя в отдельных случаях это может ухудшить различимость текста.

Цвет

Цвет является одной из важнейших характеристик человеческого восприятия. Например, одни цвета могут успокаивать человека, другие, наоборот, возбуждать. Ниже приводятся эмоциональные характеристики различных цветовых тонов [5]:

голубой — успокаивает;

красный — возбуждает;

зеленый — настраивает на добродушный и безынициативный лад;

желтый — вызывает легкомысленный настрой;

оранжевый — раскрепощает фантазию;

фиолетовый — цвет зависти, тревоги, неудовлетворенности;

коричневый — угнетает умственную активность;

черный — способствует возникновению головных болей, но снижает количество ошибок.

Серый цвет обладает одной важной особенностью — он нейтральный. Это означает, что он хорошо сочетается с любым другим цветом, с белым и черным в особенности (эти три цвета имеют один и тот же цветовой тон). С точки зрения воздействия на человека серый цвет обычно не вызывает ни положительных, ни отрицательных эмоций (это, однако, не означает, что он стимулирует продуктивную деятельность пользователя), что обуславливает использование серого цвета по умолчанию в качестве основного цвета диалоговых окон, кнопок и других элементов управления.

Следует помнить, что цвет является *субъективной* характеристикой. Цвета, которым вы отдаете предпочтение, есть особенность вашего восприятия. Иначе говоря, нельзя подстраивать цветовую гамму интерфейса под ваше собственное восприятие. Если вы используете цвет как часть интерфейса, то вы должны дать возможность пользователю установить другую цветовую гамму. Нужно также помнить о том, что некоторые пользователи могут иметь проблемы с цветовым восприятием.

Цвет следует учитывать в случае, когда необходимо обеспечить хорошую различимость текста, например, для текстовых редакторов или для систем управления. Некоторые сочетания цвета символа и фона обладают недостаточной контрастностью, другие могут приводить к раздражению пользователя. Наилучшим сочетанием считается желтый или белый цвет символов на синем фоне, а также черный цвет символов на белом фоне.

Цвет, который может быть назначен элементу управления или форме окна, может быть *системным цветом* или *цветом палитры*. Системный цвет является индексом в таблице системных цветов и предназначен для задания значения цвета при помощи цветовой схемы, являющейся одной из системных настроек. Таким образом, если в интерфейсе используются системные цвета, фактический цвет элемента управления будет определяться выбранной пользователем цветовой схемой.

Если элементу управления назначен цвет палитры, то при изменении цветовой схемы пользователем цвет элемента управления не изменится. Поэтому, если вы используете цвета палитры для задания собственной цветовой схемы интерфейса приложения, то это нужно делать для всех без исключения элементов интерфейса.

Обычно цвета палитры используют, когда для оформления интерфейса используются цветные элементы и (или) цветовые эффекты. Цветные элементы, например, цветные области, используются в двух случаях:

- для выделения важных элементов информации;
- для визуального объединения элементов.

Цветовые эффекты, например, мигающие надписи, используют в основном для привлечения внимания пользователя к событиям, происходящим во время работы программы. Цвет используется также в случаях, когда для изображения объекта используется его фактический вид.

Кнопки

Все кнопки приложения должны по возможности иметь одинаковый размер. При использовании шрифта *Tahoma* размером 8-9 пунктов вертикальный размер кнопок равен примерно 27-29 пикселей, горизонтальный — примерно 79-89 пикселей. В случае необходимости отдельные кнопки могут иметь увеличенную ширину для размещения на них длинных надписей. Сказанное относится к кнопкам без картинок.

Недопустимо использовать кнопки увеличенного размера для того, чтобы заполнить пустое пространство окна. Размещение кнопок по поверхности окна должно быть постоянным для кнопок одинакового назна-

чения, в частности, для кнопок «OK» и «Отмена». Например, при горизонтальном расположении крайней справа является кнопка «Отмена», а кнопка «OK» располагается перед ней. При вертикальном расположении кнопка «OK» располагается выше.

Горизонтальное расположение кнопок используется в случае, если окно содержит вкладки. В других случаях при горизонтальном размещении кнопок можно использовать горизонтальную черту, под которой располагаются кнопки.

Иерархия окон

Основным правилом проектирования иерархии окон является следующее — у приложения есть одно главное (первичное) окно, и, возможно, одно или несколько диалоговых (вторичных) окон, которые могут образовывать иерархию. Иначе говоря, при проектировании приложения следует избегать ситуаций, когда одно основное (первичное) окно приложения порождает другое основное (первичное) окно. Это очень распространенная ошибка программистов, впервые разрабатывающих интерфейс.

Несколько основных окон, сменяющих друг друга, могут запутать пользователя так, что он не сможет понять, в каком месте управления приложением он находится в текущий момент, что последует дальше, и как вернуться на исходную позицию. На самом деле главное окно приложения и есть исходная позиция в приложении, а все отступления от нее должны приводить только ко вторичным диалоговым окнам, которые явно приостанавливают работу главного окна.

Диалоговые окна могут образовывать иерархию, однако иерархичность диалогов должна быть оправдана. Пользователю проще работать, когда диалоги не приводят к новым диалогам, хотя в отдельных случаях это может показаться удобным.

Другой случай возникает при работе мастеров. Мастер (*wizard*) — это последовательность диалоговых окон, предназначенная для выполнения сложных действий, требующих запроса множества параметров. Диалоги мастера образуют не иерархию, а линейную последовательность, в которой пользователь может перемещаться вперед и назад. В любой момент времени работы мастера пользователь может также отказаться от его выполнения. При проектировании мастера имеет смысл использовать одну и ту же форму, рабочая область которой меняется в зависимости от шага.

Недопустимо подменять основное окно приложения диалогом. Пример такой часто встречающейся подмены приведен на рисунке 16.

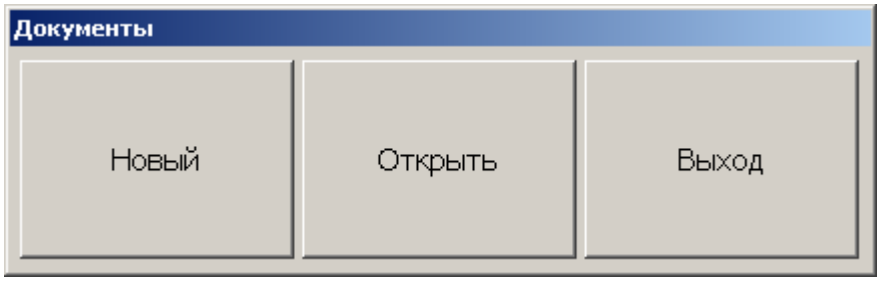

Рисунок 16 — Пример неправильного главного окна приложения

Разработчик использовал кнопки в качестве меню приложения. Нажатие кнопки «Новый» приводит к появлению окна *Microsoft Word*, нажатие кнопки «Открыть» приводит к появлению диалога выбора документа.

В этом приложении потерян смысл. Первоначально предполагалось, что приложение управляет документами пользователя, предоставляя ему услуги по их размещению, группированию в категории и поиску. Разработанный же интерфейс приложения является надстройкой, дублирующей функции *Microsoft Word*.

Одним из вариантов проектирования основного окна данного приложения является окно со списком документов пользователя, отображающим ту или иную группу документов. Группа (категория) выбирается в раскрывающемся списке над основным списком (рисунок 17).

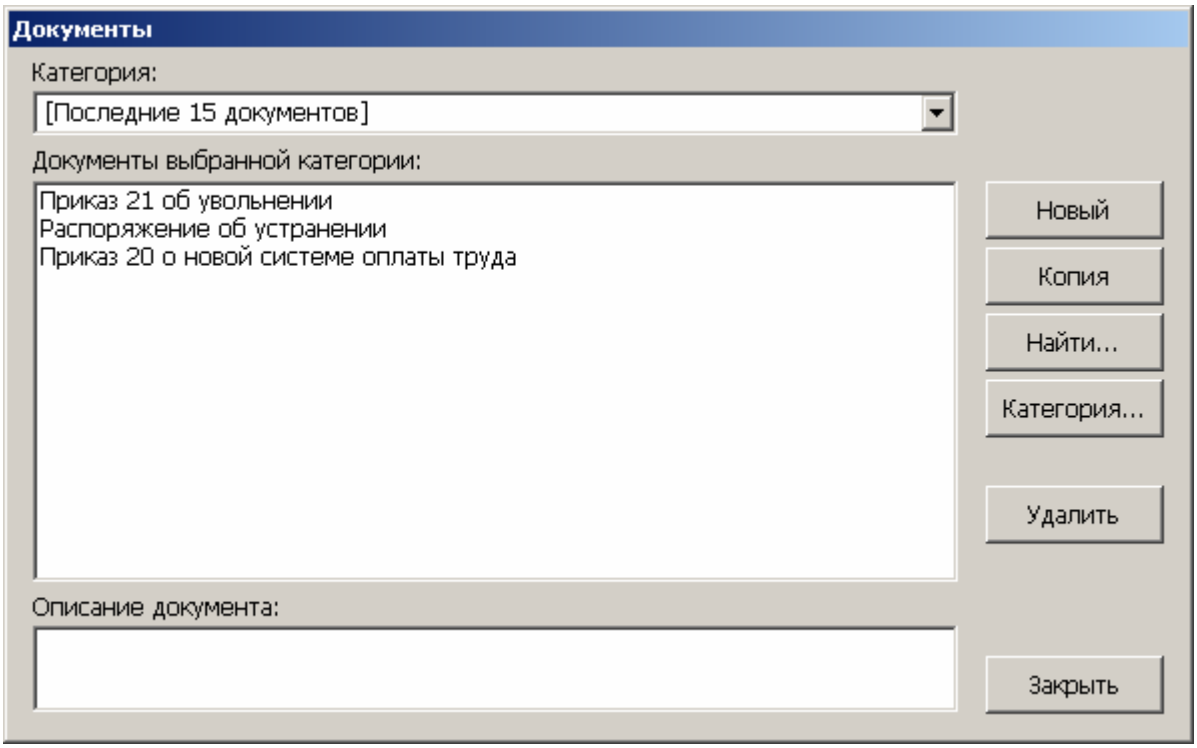

Рисунок 17 — Вариант основного окно приложения «Документы»

В списке категорий есть два специальных элемента — «[Все документы]» и «[Последние 15 документов]». Управление приложением осуществляется также при помощи кнопочного меню.

В другом случае разработчик приложения «Склад» в качестве интерфейса приложения использовал таблицы, отображающие содержимое полей таблиц нижележащей базы данных, полностью продублировав, таким образом, приложение *Microsoft Access*. В данном случае ему следовало бы разработать наглядное представление товаров на складе и диалоговые процедуры для управления ими (либо при помощи диалогов, либо при помощи дополнительных элементов управления в основном окне).

Основное меню приложения

Основное меню приложения, если есть, должно предоставлять все функции приложения. Для этого меню строится в виде иерархии, на верхнем уровне которой находятся основные пункты меню, видимые пользователю, а подуровни образуются при помощи подменю, раскрывающихся при выборе пункта.

Подпункты меню выполняют те или иные функции приложения, при этом некоторые подпункты меню раскрываются в диалоги. Такие подпункты должны иметь в троеточие в конце надписи. Например, подпункт, открывающий диалог для выбора файла, должен иметь надпись «Открыть…».

Основные пункты меню выстраиваются в определенном порядке: сначала следуют пункты «Файл», «Правка», «Вид». Далее следуют пункты, связанные с особенностями приложения, такие, как «Вставка», «Формат», «Справочник» и т.п. Заключительная последовательность пунктов меню должна содержать пункты «Параметры», «Окно», «Справка». При этом пункт «Окно» требуется в случае, если приложение имеет тип *MDI*, и предназначен для управления дочерними окнами приложения. У этого пункта меню должно установлено свойство `WindowList`.

В случае, если основное меню содержит небольшое число пунктов, допускается его реализация, например, в виде множества кнопок или картинок в основном окне.

Дополнительные элементы интерфейса

К обязательным дополнительным элементам интерфейса относятся диалог «О программе…» и справочная система. Оба элемента интерфейса вызываются через пункт «Справка» основного меню приложения. Подпункт, открывающий справку, имеет надпись «Содержание».

Диалог «О программе…» должен содержать значок приложения, название приложения, номер текущей версии, сведения о разработчике и, возможно, ссылку на сайт программы или контактный *e-mail*.

Справочная система строится на основе *HTML*-страниц и может быть скомпилирована, например, в *chm*-файл. Построение справочной системы выходит за рамки данного пособия. Отметим только, что справочная система должна быть полной, но краткой. Это означает, что в ней должны быть освещены все возможности приложения, однако при изложении способов управления приложением следует быть предельно лаконичным. Справочная система должна также описывать установку приложения, возможные ошибки, которые могут возникнуть при работе с приложением, и описание способов их устранения.

Приложению должен быть сопоставлен значок приложения и, возможно, значок документа. Значок приложения представляет *ico*-файл, содержащий маленький (16×16 пикселей), и большой (32×32 пикселя) значки, выполненные в 16-ти цветной или 256-ти цветной палитре.

Значок приложения назначается свойству **`Icon`** формы главного окна приложения. Дополнительно необходимо сделать копию значка приложения без маленького значка для диалога «О программе…».

Для того, чтобы назначить значок документа файлу документа приложения, необходимо:

а) записать значок в файл ресурсов, при помощи, например, системы программирования *MSVC++*; включить файл ресурсов к проект;

б) зарегистрировать значок в реестре *Windows*, для чего в него должна быть внесена запись

`HKCR\Имя_приложения\DefaultIcon`

которая содержит путь к исполняемому файлу, запятую и порядковый номер значка в файле ресурсов. В этой записи **`HKCR`** обозначает раздел реестра **`HKEY_CLASSES_ROOT`**, **`Имя_приложения`** — имя проекта приложения (сервера).

Чтобы приложение открывало документ при помощи функции «Открыть», в реестр дополнительно вносится запись

`HKCR\Имя_приложения\Shell\Open\Command`

которая содержит команду вида «**`путь_к_программе %1`**», а приложение при помощи функции **`Command`** определяет параметр 1, который будет спецификацией документа, и открывает его.

Методические материалы

Примерный перечень практических работ

Работа 1. Введение в автоматизацию (2 часа)

В данной работе изучаются основы управления объектом автоматизации через интерфейс *IDispatch*. В основе его использования лежит вызов метода **`Invoke`** (метода **`CallByName`** в *MSVB*) и связанная с этим упаковка параметров вызова. В работе строится приложение, которое позволяет во время работы подключиться к произвольному серверу автоматизации и вызвать его метод или свойство.

Работа 2. Сервер и контроллер автоматизации (2 часа)

В данной работе изучаются основы создания сервера и контроллера автоматизации средствами *MSVB*. Создаются простейший сервер автоматизации и его контроллер. Сервер автоматизации представляет из себя класс с одним методом. Контроллер автоматизации создает объект автоматизации и вызывает его метод. Рекомендуется самостоятельно создание сервера автоматизации, выполняющего перевод слова с английского языка на русский или наоборот.

Работа 3. Основы автоматизации MS Excel (2 часа)

В данной работе средствами *MSVB* создается контроллер автоматизации *MS Excel*. Контроллер используется для изучения объектной модели приложения. Во время выполнения работы создаются рабочие книги и расчетные листы, изучается управление ячейками.

Работа 4. Скрипты (2 часа)

В данной работе изучаются скрипты, написанные с использованием языка *Visual Basic Scripting Edition*. Исследуются хосты *wscript* и *cscript*, аргументы скриптов, вывод сообщений пользователю. Изучается файловая объектная модель *Microsoft Scripting Runtime*.

Работа 5. Основы автоматизации MS Word (2 часа)

В данной работе изучаются основы автоматизации *MS Word*. Исследуется управление документами, параграфами и выделением, а также получение объекта **`Range`**. Изучаются коллекции **`Words`** и **`Characters`**. Изучается поиск и замена при помощи метода **`Find`**.

Работа 6. Автоматизация Excel и Word (2 часа)

В данной работе изучается слияние документов, выполняемое при помощи *MS Word* и *MS Excel*. Результат слияния выводится в файл. Для выполнения работы используется запись макросов. Формируется скрипт для автоматической рассылки электронных писем с использованием источника данных слияния.

Работа 7. Проверка правописания (2 часа)

В данной работе разрабатывается простейшее приложение типа «Блокнот», позволяющее выполнять проверку текста при помощи средств *MS Word* для проверки правописания и грамматики.

Работа 8. Основы создания сервера-приложения (4—8 часов)

В данной работе изучаются основы создания простого автоматизированного приложения «Рецепты». Разрабатывается класс для управления базой данных. Создаются классы объектной модели. Простейшими средствами формируется интерфейс приложения диалогового типа. Объем выполняемой работы зависит от количества выделяемых учебных часов.

Примерные темы курсовых проектов

На курсовое проектирование выносится создание автоматизированного приложения-сервера. Данные приложения сохраняются в файле базы данных *Microsoft Access*. Примерный перечень тем проектов следующий:

1. «Документы секретаря»
2. «Документы кафедры»
3. «Дневник студента»
4. «Расписание студента»
5. «Дневник преподавателя»
6. «Расписание преподавателя»
7. «Учет успеваемости»
8. «Документы преподавателя»
9. «Составитель расписания»
10. «Библиотека»
11. «Фотоальбом»
12. «Справочник литературы»
13. «Домашний бюджет»
14. «Контакты»
15. «Блок-схемы»
16. «Схема сети»

Список использованных источников

1. Трельсен Э. Модель COM и применение ATL 3.0: Пер. с англ. — СПб.: BHV — Санкт-Петербург, 2000. — 928 с.: ил.

2. Kraig Brockschmidt. Inside OLE — MSDN, January 2001 — Books.

3. Технология разработки программного обеспечения: Учебник для вузов. 3-е изд. / С.А. Орлов. — СПб.: Питер, 2007. — 527 с.: ил.

4. Раскин Д. Интерфейс: новые направления в проектировании компьютерных систем. — Пер. с англ. — СПб: Символ-Плюс, 2003. — 272 с., ил.

5. Гультяев А.К., Машин В.А. Проектирование и дизайн пользовательского интерфейса. СПб.: КОРОНА принт, 2000. — 352 с.

6. Official Guidelines for User Interface Developers and Designers. MSDN, October 2001 — Book Excerpts — The Windows User Interface.

7. Microsoft Inductive User Interface Guidelines. Microsoft Corporation. MSDN, October 2001 — Technical Articles — Windows Platform — User Interface.

Приложение А — Функции для типа VARIANT

Основные функции

HRESULT VariantInit(VARIANTARG*);

Инициализирует **VARIANT**, устанавливая в поле **vt** значение **VT_EMPTY**.

HRESULT VariantClear(VARIANTARG*);

Освобождает память, занимаемую хранимым значением (для строк, массивов и объектов), и устанавливает в поле **vt** значение **VT_EMPTY**.

HRESULT VariantCopy(VARIANTARG* приемник, VARIANTARG* источник);

Копирует **VARIANT**-источник в **VARIANT**-приемник. Перед выполнением операции освобождает приемник.

Возвращаемые значения:

S_OK — преобразование успешно;

DISP_E_ARRAYISLOCKED — массив заблокирован;

DISP_E_BADVARTYPE — недопустимый (не совместимый) тип;

E_OUTOFMEMORY — недостаточно памяти для копии;

E_INVALIDARG — один из аргументов неправильный.

HRESULT VariantCopyInd(VARIANTARG* приемник, VARIANTARG* источник);

Копирует **VARIANT**-источник в **VARIANT**-приемник. Если источник был указателем, приемник будет значением. Перед выполнением операции освобождает приемник.

Возвращаемые значения: см. предыдущую функцию.

HRESULT VariantChangeType(VARIANTARG* приемник,VARIANTARG* источник, unsigned short преобразование, VARTYPE новый_тип);

Выполняет приведение типа источника к типу, указанному параметром **новый_тип**, и записывает новое значение в приемник. Параметр **преобразование** может иметь одно из следующих значений:

VARIANT_NOVALUEPROP — использовать в преобразовании объект вместо его значения по умолчанию;

VARIANT_ALPHABOOL — преобразовать логическое значение в строку;

VARIANT_NOUSEROVERRIDE — при преобразовании, в котором участвует строковое значение, не изменять идентификатор локали;

VARIANT_LOCALBOOL — при преобразовании строкового значения в логическое и наоборот использовать текущую локаль.

0 — выполнить преобразование по умолчанию.

Возвращаемые значения:

S_OK — преобразование успешно;

DISP_E_ARRAYISLOCKED — массив заблокирован;

DISP_E_BADVARTYPE — недопустимый (не совместимый) тип;

E_INVALIDARG — один из аргументов неправильный.

```
HRESULT VariantChangeTypeEx(VARIANTARG* приемник,
    VARIANTARG* источник, LCID lcid,
    unsigned short преобразование, VARTYPE новый_тип);
```

Отличается от предыдущей функции идентификатором локали, которая учитывается при преобразованиях, в которых участвуют строковые, числовые и логические значения. Например, число «1,25» может быть приведено к строке «1,25» или «1.25» в зависимости от указанной локали.

Возвращаемые значения:

S_OK — преобразование успешно;

DISP_E_ARRAYISLOCKED — массив заблокирован;

DISP_E_BADVARTYPE — недопустимый (не совместимый) тип;

E_INVALIDARG — один из аргументов неправильный.

Математические функции

Следующие функции принимают три аргумента типа **VARIANT***: первый указывает на левый аргумент выражения, второй — на правый, третий принимает результат.

VarAdd — Вычисляет сумму.

VarSub — Вычисляет разность.

VarMul — Вычисляет произведение.

VarDiv — Вычисляет результат вещественного деления.

VarIdiv — Вычисляет результат целочисленного деления.

VarCat — Выполняет конкатенацию. Результат зависит от исходных типов. Во всех случаях функция поступает в высшей степени разумно.

VarAnd — Вычисляет результат операции **AND**.

VarOr — Вычисляет результат операции **OR**.

VarXor — Вычисляет результат операции **XOR**.

VarMod — Вычисляет результат операции **MOD**.

VarPow — Возводит левый аргумент в степень — правый аргумент.

Следующие функции принимают два аргумента типа **VARIANT***: первый указывает на исходное значение, второй — на результат.

VarNot — Вычисляет результат операции **NOT**.

VarAbs — Вычисляет абсолютное значение.

VarFix — Вычисляет целую часть числа.

VarInt — Вычисляет целую часть числа.

VarNeg — Вычисляет отрицательное значение исходного числа.

Следующие функции принимают разное количество аргументов:

```
VarRound(LPVARIANT pvarIn, INT cDecimals, LPVARIANT pvarResult);
```

Округляет число, заданное аргументом **pvarIn**, до указанного аргументом **cDecimals** количества знаков.

```
VarCmp(LPVARIANT pvarLeft, LPVARIANT pvarRight, LCID lcid, ULONG
dwFlags);
```

Сравнивает два значения. Параметр **lcid** задает локаль. Параметр **dwFlags** определяет, как выполнять сравнение. Допустимые значения:

NORM_IGNORECASE (0x1) — игнорировать регистр;

NORM_IGNORENONSPACE (0x2) — игнорировать непробельные символы;

NORM_IGNORESYMBOLS (0x4) — игнорировать символы;

NORM_IGNOREWIDTH (0x8) — игнорировать длину строки;

0 — выполнить сравнение по умолчанию.

Возвращаемые значения:

VARCMP_LT (0) — левый аргумент меньше правого;

VARCMP_EQ (1) — аргументы равны;

VARCMP_GT (2) — левый аргумент больше правого;

VARCMP_NULL (3) — один или оба аргумента содержат значение **NULL**.

*Функции для типа **Currency***

Следующие функции принимают три аргумента: первый указывает на левый аргумент выражения, второй — на правый, третий принимает результат.

VarCyAdd(CY, CY, LPCY); — Вычисляет сумму.

VarCyMul(CY, CY, LPCY); — Вычисляет произведение.

VarCyMulI4(CY, Long, LPCY); — Вычисляет произведение типа **Currency** на тип **Long**.

VarCySub(CY, CY, LPCY) — Вычисляет разность.

Следующие функции принимают два аргумента: первый указывает на исходное значение, второй — на результат.

VarCyAbs(CY, LPCY); — Вычисляет абсолютное значение.

VarCyFix(CY, LPCY); — Вычисляет целую часть значения.

VarCyInt(CY, LPCY); — Вычисляет целую часть значения.

VarCyNeg(CY, LPCY); — Вычисляет отрицательное значение.

Следующая функция округляет значения до указанного количества знаков после запятой, указанное во втором аргументе:

```
VarCyRound(CY, INT, LPCY);
```

Следующая функция сравнивает два значения типа **Currency**:

```
VarCyCmp(CY, CY);
```

Возвращаемые значения соответствуют возвращаемым значениям функции **VarCmp**.

Следующая функция сравнивает значение типа **Currency** со значением типа **Double**:

```
VarCyCmpR8(CY, double);
```

Возвращаемые значения соответствуют возвращаемым значениям функции **VarCmp**.

BSTR SysAllocString(const OLECHAR *);

Резервирует память и копирует в нее строку, указанную аргументом. Возвращает указатель на строку в случае успеха или **NULL**, если не удалось выделить память или аргумент равен **NULL**.

INT SysReAllocString(BSTR FAR *, const OLECHAR FAR);

Резервирует новую память для **BSTR** и копирует в нее строку, указанную вторым аргументом. Возвращает **TRUE** в случае успеха или **FALSE** в случае неудачи.

BSTR SysAllocStringLen(const OLECHAR *, unsigned int);

Резервирует память, копирует в нее **cch** символов строки, указанной первым аргументом и добавляет нулевой символ. Возвращает указатель на строку в случае успеха или **NULL**, если не удалось выделить память.

BSTR SysAllocStringByteLen(LPCSTR, unsigned int);

Резервирует память, копирует в нее **len** символов строки *ANSI*, указанной первым аргументом и добавляет нулевой символ. Возвращает указатель на строку в случае успеха или **NULL**, если не удалось выделить память. Никаких преобразований символов не производится. Функция может быть использована для создания массива двоичных данных, включающих нулевые символы.

```
BSTR SysReAllocStringLen(BSTR FAR *, const OLECHAR FAR *,
    unsigned int cch);
```

Резервирует память, копирует в нее **cch** символов строки, указанной вторым аргументом, добавляет нулевой символ и освобождает строку, указанную вторым аргументом. Возвращает **TRUE** в случае успеха или **FALSE** в случае неудачи.

VOID SysFreeString(BSTR);

Освобождает память, выделенную функциями типа **SysAllocXXXX** и **SysReAllocXXXX**.

UINT SysStringLen(BSTR);

Возвращает длину строки, исключая завершающий нулевой символ. Нулевые символы внутри строки учитываются.

UINT SysStringByteLen(BSTR);

Возвращает длину строки в байтах (используется только в 32-разрядных приложениях).

Приложение В — Классы стандартных диалогов

Классы используют следующие объявления, типы и перечисления, объявляемые в стандартном модуле.

Объявления системных функций:

```
Public Declare Function GetOpenFileName Lib "comdlg32.dll" Alias
"GetOpenFileNameA" (OFN As OFNType) As Boolean
Public Declare Function GetSaveFileName Lib "comdlg32.dll" Alias
"GetSaveFileNameA" (OFN As OFNType) As Boolean
```

Структура данных диалога:

```
Public Type OFNType
    StructSize As Long
    Owner As Long
    hInstance As Long
    Filter As String
    CustomFilter As String
    MaxCustFilter As Long
    FilterIndex As Long
    File As String
    MaxFile As Long
    FileTitle As String
    MaxFileTitle As Long
    InitDir As String
    Title As String
    flags As Long
    FileOffset As Integer
    FileExtention As Integer
    DefaultExt As String
    CustomData As Long
    Hook As Long
    TemlplateName As String
End Type
```

Перечисление констант стандартного диалога

```
Public Enum OFNGeneral
    OFN_READONLY = &H1
    OFN_HIDEREADONLY = &H4
    OFN_OVERWRITEPROMPT = &H2
    OFN_PATHMUSTEXIST = &H800
    OFN_FILEMUSTEXIST = &H1000
    OFN_EXPLORER = &H80000
    OFN_LONGNAMES = &H200000
End Enum
```

Основной текст классов **DialogOpen** и **DialogSave**

```
Private OFN As OFNType
```

Конструктор класса **DialogOpen**

```
Private Sub Class_Initialize()
    With OFN
        .StructSize = Len(OFN)
        .Filter = csFilterAllFiles
        .Title = "Открытие файла"
        .Owner = 0
        .File = String(255, 0)
        .MaxFile = 255
        .FileTitle = String(255, 0)
        .MaxFileTitle = 255
        .FilterIndex = 0
        .flags=OFN_HIDEREADONLY+OFN_PATHMUSTEXIST+OFN_FILEMUSTEXIST
        .InitDir = "C:\"
        .hInstance = 0
        .CustomFilter = String(255, 0)
        .MaxCustFilter = 255
        .Hook = 0
    End With
End Sub
```

Конструктор класса **DialogSave**

```
Private Sub Class_Initialize()
    With OFN
        .StructSize = Len(OFN)
        .Filter = csFilterAllFiles
        .Title = "Сохранение файла"
        .Owner = 0
        .File = String(255, 0)
        .MaxFile = 255
        .FileTitle = String(255, 0)
        .MaxFileTitle = 255
        .FilterIndex = 0
        .flags = OFN_PATHMUSTEXIST+OFN_OVERWRITEPROMPT
        .InitDir = "C:\"
        .hInstance = 0
        .CustomFilter = String(255, 0)
        .MaxCustFilter = 255
        .Hook = 0
    End With
End Sub
```

Продолжение основного текста классов

```
' Возвращает/устанавливает расширение файла по умолчанию.
Public Property Get DefaultExt() As String
    DefaultExt = OFN.DefaultExt
End Property
Public Property Let DefaultExt(ByVal NewValue As String)
    OFN.DefaultExt = NewValue
End Property

' Возвращает/устанавливает начальный каталог.
Public Property Get InitDir() As String
    InitDir = OFN.InitDir
End Property
Public Property Let InitDir(ByVal NewValue As String)
    OFN.InitDir = NewValue
End Property

' Возвращает/устанавливает путь к файлу.
Public Property Get File() As String
    File = Left(OFN.File, InStr(1, OFN.File, Chr(0)) - 1)
End Property
Public Property Let File(ByVal NewValue As String)
    OFN.File = NewValue & String(255 - Len(NewValue), 0)
End Property

' Возвращает/устанавливает имя файла.
Public Property Get FileTitle() As String
    FileTitle=Left(OFN.FileTitle, InStr(1, OFN.FileTitle, Chr(0))-1)
End Property
Public Property Let FileTitle(ByVal NewValue As String)
    OFN.FileTitle = NewValue & String(255 - Len(NewValue), 0)
End Property

' Возвращает/устанавливает маски файлов.
Public Property Get Filter() As String
    Filter = OFN.Filter
End Property
Public Property Let Filter(ByVal NewValue As String)
    OFN.Filter = NewValue
End Property

' Возвращает/устанавливает номер текущей маски файла.
Public Property Get FilterIndex() As Integer
    Filter = OFN.FilterIndex
End Property
```

```
Public Property Let FilterIndex(ByVal NewValue As Integer)
    OFN.FilterIndex = NewValue
End Property
' Устанавливает владельца диалога.
Public Property Let Owner(ByVal NewValue As Long)
    OFN.Owner = NewValue
End Property
' Возвращает/устанавливает заголовок диалога.
Public Property Get Title() As String
    Title = OFN.Title
End Property
Public Property Let Title(ByVal NewValue As String)
    OFN.Title = NewValue
End Property
```

Основной метод класса `DialogOpen`

```
' Возвращает имя открываемого файла.
Public Function GetFileName() As Boolean
    If (GetOpenFileName(OFN) = False) Then
        File = ""
        Exit Function
    End If
    InitDir = Left(File, Len(File) - Len(FileTitle))
    GetFileName = True
End Function
```

Основной метод класса `DialogSave`

```
' Возвращает имя открываемого файла.
Public Function GetFileName() As Boolean
    If (GetSaveFileName(OFN) = False) Then
        File = ""
        Exit Function
    End If
    InitDir = Left(File, Len(File) - Len(FileTitle))
    GetFileName = True
End Function
```

Владимир Вадимович Пономарев
Проектирование автоматизированных программных систем.
Учебно-практическое пособие.
Отпечатано с готового оригинал-макета.
Издательство ОТИ МИФИ, 2009
Тираж 40 экз.
