Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Институт повышения квалификации
и переподготовки

Кафедра «Информатика»

Н. В. Самовендюк

ТЕХНОЛОГИИ КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ

ПОСОБИЕ
по одноименному курсу
для слушателей специальности 1-40 01 73
«Программное обеспечение информационных систем»
заочной формы обучения

Гомель 2016

УДК 004.43(075.8)
ББК 32.973-018.2я73
С17

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 5 от 28.12.2015 г.*)

Рецензент: доц. каф. «Информационные технологии» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *В. И. Токочаков*

Самовендюк, Н. В.

С17 Технологии компонентного программирования : пособие по одноим. курсу для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заоч. формы обучения / Н. В. Самовендюк. – Гомель : ГГТУ им. П. О. Сухого, 2016. – 123 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Содержит основной теоретический материал дисциплины «Технологии компонентного программирования». Рассмотрены основные понятия компонентного программирования, программирования на основе интерфейсов, программной поддержки модели составных объектов и создания компонентов, использования библиотек для создания компонентов.

Для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заочной формы обучения ИПКиП.

УДК 004.43(075.8)
ББК 32.973-018.2я73

ПРЕДИСЛОВИЕ

Пособие по дисциплине «Технологии компонентного программирования» разработано для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заочной формы обучения ИПК и ПК. Оно предусматривает изучение основных понятий компонентного программирования, так как понятие программного компонента (software component) является одним из ключевых в современной инженерии программного обеспечения. Технологии компонентного программирования более известны как технологии COM (Component Object Model),модель компонентных объектов. Они широко применяются во всех версиях Windows, но могут использоваться и в других операционных системах. На основе COM создано множество других технологий: Microsoft OLE Automation, ActiveX, DCOM, COM+, а также XPCOM.

Целью данного пособия является получение студентами базовых знаний COM-технологий, изучение вопросов использования интерфейсов, взаимодействия клиентских приложений с Com-серверами, реализованных в виде динамически загружаемых библиотек или во внешней программе, управления Com-объектами во время выполнения программы.

Большое внимание уделено созданию собственных Com-серверов, как серверов низкого уровня, так и серверов с библиотекой типов.

Отдельно рассматривается технология OLE Automation. На практических примерах показывается возможность создания контроллеров автоматизации для популярных серверов автоматизации Microsoft Word и Microsoft Excel.

Все примеры написаны на объектно-ориентированном языке Delphi.

1 Основы компонентных технологий

Компонентный подход к проектированию и реализации программных систем и комплексов является развитием объектно-ориентированного подхода и чаще всего используется при разработке крупных и распределенных программных систем. С точки зрения компонентного подхода программная система – это набор компонентов с четко определенным интерфейсом. В отличие от других подходов программной инженерии, изменения в систему вносятся путем создания новых компонентов или изменения старых, а не путем рефакторинга существующего кода.

Основой компонентных технологий является программный компонент – это автономный элемент программного обеспечения, предназначенный для многократного использования, который может распространяться для использования в других программах в виде скомпилированного кода.

1.1 Компоненты

Обычно приложение состоит из одного монолитного двоичного файла. После того, как приложение сгенерировано компилятором, он остается неизменным — пока не будет скомпилирована и поставлена пользователю новая версия. Чтобы учесть изменения в операционных системах, аппаратуре и желаниях пользователей, приходится ждать перекомпиляции приложения.

При современных темпах развития индустрии программирования приложениям нельзя оставаться застывшими. Необходимо найти возможность модернизации программ, поставленных пользователям. Решение состоит в том, чтобы разбить монолитное приложение на отдельные части, или компоненты (рисунок 1.1).

Разбиение монолитного приложения (слева) на компоненты (справа) облегчает адаптацию. По мере развития технологии компоненты, составляющие приложение, могут заменяться новыми (рисунок 1.2).

Рисунок 1.1

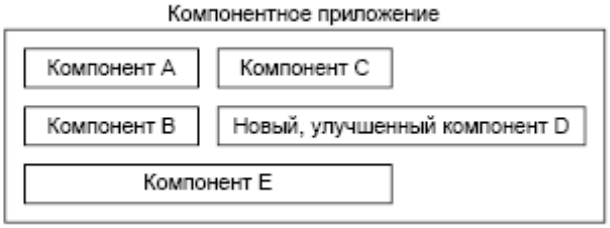

Рисунок 1.2

Приложение более не является статичным. Оно постепенно эволюционирует с заменой старых компонентов новыми. Из существующих компонентов легко создать и абсолютно новые приложения.

Традиционно приложение состоит из отдельных файлов, модулей или классов, которые компилируются и компонуются в единое целое. Разработка приложений из компонентов — так называемых приложений **компонентной архитектуры** — происходит совершенно иначе. Компонент подобен **миниприложению,** он поставляется пользователю как двоичный код, скомпилированный и готовый к использованию. Единого целого больше нет. Его место занимают специализированные компоненты, которые подключаются во время выполнения к другим компонентам, формируя приложение. Модификация или расширение приложения сводится просто к замене одного из составляющих его компонентов новой версией.

Для того, чтобы разбить монолитное приложение на компоненты, необходим мощный инструмент. Такой инструмент называется **COM**. **COM — модель компонентных объектов (Component Object Model)**.

1.2 Преимущества использования компонентов

Преимущества использования компонентов непосредственно вытекают из способности последних подключаться к приложению и отключаться от него.

Использование компонент при разработке программного обеспечения, кроме удобства и гибкости при модернизации существующих приложений, дает ряд других преимуществ. Они связаны с адаптацией приложений к нуждам пользователя, библиотеками компонентов и распределенными компонентами.

Компонентные архитектуры хорошо приспособлены для адаптации, так как любой компонент можно заменить другим, более

соответствующим потребностям пользователя. Пользователи часто хотят подстроить приложения к своим нуждам, чтобы приложение работало так, как они привыкли.

Предположим, что у нас есть компоненты на основе редакторов ***vi*** и **Emacs**. Как видно из рисунка 1.3, пользователь 1 может настроить приложение на использование ***vi,*** тогда как пользователь 2 — предпочесть **Emacs**. Приложения можно легко настраивать, добавляя новые компоненты или заменяя имеющиеся.

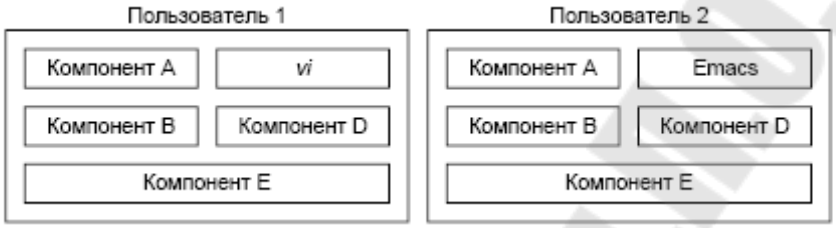

Рисунок 1.3

Использование библиотеки компонентов предоставляет возможность быстрой разработки приложений. Выбирая компоненты из библиотеки, можно составлять из них, как из деталей конструктора, цельные приложения (рисунок 1.4).

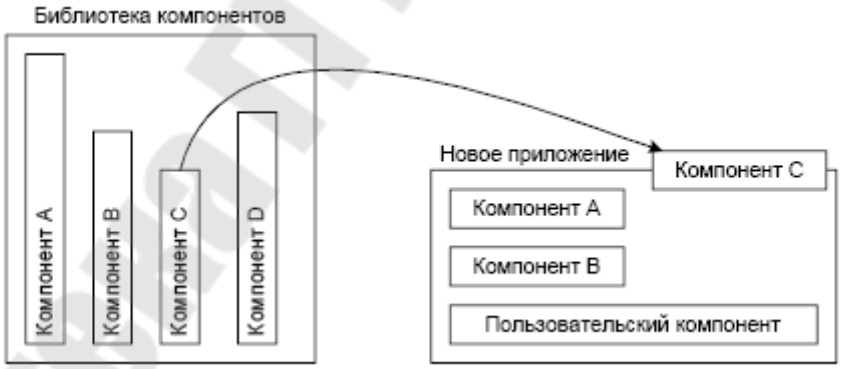

Рисунок 1.4

Сборка приложений из стандартных блоков давно была заветной мечтой программистов. Этот процесс уже начался с созданием управляющих элементов ActiveX (ранее известных как управляющие элементы OLE).

Программисты на Visual Basic, Delphi, C, C++, C# и Java могут воспользоваться управляющими элементами ActiveX для ускорения разработки своих приложений и страниц Web. Конечно, каждому

приложению по-прежнему будут нужны и некоторые специализированные компоненты, но в основном можно будет обойтись стандартными.

Компонентная архитектура помогает упростить процесс разработки распределенных приложений. Создать из обычного приложения распределенное легче, если это обычное приложение состоит из компонентов. Во-первых, оно уже разделено на функциональные части, которые могут располагаться вдали друг от друга. Во-вторых, поскольку компоненты заменяемы, вместо некоторого компонента можно подставить другой, единственной задачей которого будет обеспечивать связь с удаленным компонентом.

Например, на рисунке 1.5 компонент C и компонент D расположены в сети на двух удаленных машинах. На локальной машине они заменяются двумя новыми компонентами, переадресовщиками C и D. Последние переправляют запросы от других компонентов к удаленным C и D по сети. Для приложения на локальной машине неважно, что настоящие компоненты C и D находятся где-то в другом месте. Точно так же для самих удаленных компонентов не имеет значения их расположение. При наличии подходящих переадресующих компонентов приложение может совершенно игнорировать фактическое местоположение своих частей.

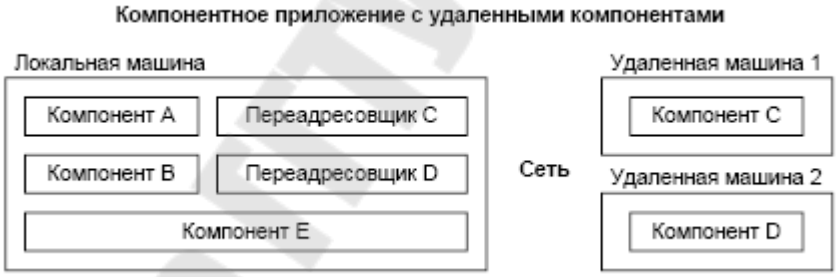

Рисунок 1.5 Расположение компонентов на удаленных машинах в сети

1.3 Требования, предъявляемые к компонентам

Поскольку компоненты подключаются к приложению и отключаются от него во время работы, они должны удовлетворять двум требованиям. Во-первых, они должны компоноваться динамически. Во-вторых, они должны скрывать (или ***инкапсулировать***) детали своей реализации.

Поддержка подключения или замены компонента во время выполнения требует **динамической компоновки**.

Чтобы понять, как это важно, лучше всего представить себе приложение, построенное из компонентов, которые не могут объединяться

во время выполнения. Если Вы захотите изменить один из компонентов такой системы, Вам придется статически перекомпоновать или перекомпилировать программу и заново разослать ее пользователям. В конце концов, нельзя требовать от конечных пользователей, чтобы они перекомпилировали приложение самостоятельно. Даже если они знают, как это сделать, у них скорее всего нет компоновщика — либо вообще, либо конкретного, необходимого. Приложение, собранное из компонентов, которые надо статически перекомпоновывать каждый раз при замене одного из них, эквивалентно приложению-монолиту.

Динамическая компоновка требует **инкапсуляции**. Чтобы сформировать приложение, компоненты подключаются друг к другу. Если Вы хотите заменить один из компонентов, надо отключить от системы старый и подсоединить новый. Новый компонент должен подсоединяться тем же способом, что и старый, иначе компоненты придется переписывать, перекомпилировать и перекомпоновывать.

Чтобы понять, как это связано с инкапсуляцией, нам необходимо определить некоторые термины. Программа или компонент, использующие другой компонент, называется ***клиентом* (*client*).** Клиент подсоединяется к компоненту через ***интерфейс* (*interface*)**. Если компонент изменяется без изменения интерфейса, то изменений в клиенте не потребуется. Аналогично, если клиент изменяется без изменения интерфейса, то нет необходимости изменять компонент. Однако если изменение либо клиента, либо компонента вызывает изменение интерфейса, то и другую сторону интерфейса также необходимо изменить.

Таким образом, для того, чтобы воспользоваться преимуществами динамической компоновки, компоненты и клиенты должны стараться не изменять свои интерфейсы. Они должны быть инкапсулирующими. Детали реализации клиента и компонента не должны отражаться в интерфейсе. Чем надежнее интерфейс изолирован от реализации, тем менее вероятно, что он изменится при модификации клиента и компонента. Если интерфейс не изменяется, то изменение компонента оказывает лишь незначительное влияние на приложение в целом.

Необходимость изоляции клиента от деталей реализации накладывает на компоненты ряд важных ограничений:

1. Компонент должен скрывать используемый язык программирования. Любой клиент должен иметь возможность использовать компонент, независимо от языков программирования, на которых написаны тот или другой. Раскрытие языка реализации создает новые зависимости между клиентом и компонентом.

2. Компоненты должны распространяться в двоичной форме. Действительно, поскольку они должны скрывать язык реализации, их

необходимо поставлять уже скомпилированными, скомпонованными и готовыми к использованию.

3. Должна быть возможность модернизировать компоненты, не затрагивая уже существующих пользователей. Новые версии компонента должны разработать как с новыми, так и со старыми клиентами.

4. Должна быть возможность прозрачно перемещать компоненты в сети. Необходимо, чтобы компонент и использующая его программа могли выполняться внутри одного процесса, в разных процессах или на разных машинах. Клиент должен рассматривать удаленный компонент так же как и локальный. Если бы с удаленными компонентами надо было работать иначе, чем с локальными, то потребовалось бы перекомпилировать клиент всякий раз, когда локальный компонент перемещается в другое место сети.

2 Технология COM

COM — это спецификация. Она указывает, как создавать динамически взаимозаменяемые компоненты. COM определяет стандарт, которому должны следовать компоненты и клиенты, чтобы гарантировать возможность совместной работы. Стандарты важны для компонентных архитектур так же, как и для любой системы с взаимозаменяемыми частями.

Спецификация COM (COM Specification) — это документ, который устанавливает стандарт для компонентной архитектуры. Спецификация COM содержится на сайте Microsoft.

Компоненты COM состоят из исполняемого кода, распространяемого в виде динамически компонуемых библиотек (DLL) или EXE-файлов Win32. Компоненты, написанные в соответствии со стандартом COM, удовлетворяют всем требованиям компонентной архитектуры.

Компоненты COM подключаются друг к другу динамически. Для этой цели COM использует DLL.

Компоненты COM полностью независимы от языка программирования. Они могут быть разработаны с помощью практически любого процедурного языка, включая Ada, C, Java, Modula-3, Oberon и Pascal.

Компоненты COM — отличный способ предоставления объектно-ориентированных API или сервисов другим приложениям. Они прекрасно подходят для создания библиотек, не зависящих от языка программирования компонентов, из которых можно быстро строить новые приложения.

2.1 Краткая история технологии COM

В Windows 3.1 и более ранних версиях основным средством обмена данных между программами была технология **DDE - Dynamic Data Exchange** (динамический обмен данными). На этой технологии основывалась технология **OLE - Object Linking and Embedding** (связывание и внедрение объектов). **OLE** позволяет делать документы одного приложения частью документов другого приложения. Таким образом, пользователь получил возможность использовать функции многих различных программ для редактирования одного документа.

В основе **DDE** лежит обмен сообщениями между окнами Windows. Подобный механизм затрудняет распараллеливание процессов и обмен данными через сеть между приложениями, работающими на разных

компьютерах (существует сетевое расширение **DDE - NetDDE**, но оно работает медленно и не всегда устойчиво).

Начиная с 1993-его года, в Windows NT 3.51 появилась технология **OLE 2** - дальнейшее развитие OLE. **OLE 2** дополнительно содержит в себе технологии **ActiveX**, **Automation** (первоначально называвшаяся **OLE Automation**) и другие расширения, далеко выходящие за рамки связывания и внедрения объектов, поэтому фирма Microsoft с выходом **OLE 2** объявила, что слово **"OLE"** больше не является аббревиатурой, это просто термин, не имеющий расшифровки.

Технология **DDE** была недостаточной для поддержки **OLE 2**, поэтому специально под неё была создана новая технология взаимодействия между программами - **COM (Component Object Model, модель компонентных объектов). COM** оказалась очень удачной технологией, поэтому, начиная с Windows 95, **DDE** была объявлена устаревшей, а основной технологией обмена данными в системе стала технология **COM**.

С появлением в 1996 г. Windows NT 4.0 технология **COM** была существенно расширена. Добавилась поддержка различных нитевых моделей и возможность организации взаимодействия программ через локальную сеть. Последнее расширение получило название **DCOM** - **Distributed COM** (распределённая модель компонентных объектов).

Модель COM/DCOM построена по принципу клиент-сервер. Сильно упрощая, можно сказать, что сервер экспортирует функции, которые клиент может вызывать, вынуждая сервер выполнить то или иное действие. Если взаимодействие между клиентом и сервером подразумевает обмен данными, эти данные передаются в качестве параметров функций. При необходимости клиент также может экспортировать функции, которые могут быть вызваны сервером.

На данный момент DCOM является межплатформенной технологией. Существуют средства для поддержки DCOM в различных UNIX-системах (в том числе Linux), Solaris, MacOS, VxWorks. Сетевой протокол DCOM опубликован в открытых источниках, поэтому реализовать его, в принципе, возможно на любой системе.

2.2 Базовые понятия COM

2.2.1 Идентификация объектов COM. GUID

Для идентификации однотипных объектов обычно используются уникальные имена или номера. Однако такой метод не может гарантировать уникальность в ситуации, когда разработчики по всему миру независимо друг от друга создают объекты и присваивают им

идентификаторы. Поэтому в качестве идентификатора в COM используется **GUID - Globally Unique Identifier** (глобально уникальный идентификатор). Иногда его также называют **UUID - Universally Unique Identifier** (вселенский уникальный идентификатор).

GUID представляет собой последовательность из 128 бит (16 байт), алгоритм генерации которой обеспечивает уникальность получившейся последовательности. При генерации используется текущее время и номер сетевой карты компьютера, а также случайные числа. Алгоритм гарантирует уникальность получающихся значений примерно до 3400 года. Если на компьютере нет сетевой карты, уникальность значения не гарантируется.

В строковом виде **GUID** представляют в виде

xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx,

заключённой в фигурные скобки, в которой каждый символ "x" обозначает шестнадцатеричную цифру. Пример строковой записи **GUID**:

{6F9619FF-8B86-D011-B42D-00C04FC964FF}

Хотя GUID очень интенсивно используется в COM, он не является составной частью этой технологии. GUID используется везде, где необходим уникальный идентификатор. Например, тип **UniqueIdentifier** в **SQL Server** - это **GUID**.

В дальнейшем мы будем сталкиваться с аббревиатурами **CLSID** (class identifier, идентификатор класса), **IID** (interface identifier, идентификатор интерфейса), **AppID** (application identifier, идентификатор приложения), **LibID** (library identifier, идентификатор библиотеки) и т. п. Всё это - разновидности **GUID**. Внутренне они совершенно одинаковы, а название каждого из этих идентификаторов лишь указывает, для идентификации какого объекта используется данный GUID.

2.2.2 Тип HRESULT

Тип **HRESULT** является одним из средств контроля ошибок в COM/DCOM. Этот тип представляет собой 32-битное число, в котором кодируется результат операции. Старший бит этого числа равен 1, если была ошибка, и 0, если всё прошло нормально. Следующие 4 бита зарезервированы для дальнейшего использования. Следующие 11 бит показывают, где возникла ошибка (это значение обычно называется **facility code**, что можно приблизительно перевести как код устройства, если подразумевать под устройством не только аппаратные, но и логические устройства). Младшие 16 бит кодируют собственно ошибку.

Для расшифровки значения типа HRESULT можно использовать утилиту **ErrLook.exe** из пакета **Visual Studio**. Более удобным средством расшифровки является утилита **Decode**, входящая в состав **MSDN**. Утилита **Decode** включена в **MSDN** как пример в исходных кодах на C++, поэтому, не имея компилятора для этого языка, воспользоваться данной утилитой не удастся.

В модели **COM** рекомендуется, чтобы функции, экспортируемые сервером и клиентом, возвращали результат типа **HRESULT**, по которому можно судить об успешном или неуспешном выполнении функции. Даже если логика работы функции подразумевает возвращение какого-то результата, его рекомендуется возвращать через параметры, передаваемые по ссылке, а тип самой функции должен быть **HRESULT**. В модели **DCOM** использование **HRESULT** - это уже не рекомендация, а обязательное требование. Некоторые функции стандартных интерфейсов, доставшиеся **DCOM** в наследство от **COM** (например, функции **AddRef** и **Release** интерфейса **IUnknown**), тем не менее, имеют тип, отличный от HRESULT. Но это допускается только для стандартных интерфейсов в целях совместимости, а все функции новых серверов должны иметь тип **HRESULT**.

Существуют предопределённые константы типа **HRESULT**, часть которых показана в таблице 1. Значения констант могут быть разными на разных платформах, поэтому в целях совместимости лучше пользоваться их символьными именами, а не значениями.

Таблица 1

Константа	Описание
S_OK	Успешное *завершение* операции
S_FALSE	Успешное завершение операции. Отличается от S_OK тем, что подразумевает какую-то особенность при выполнении функции. Использование S_FALSE не регламентируется строго, в каких случаях будет использовано значение S_OK, а в каких - S_FALSE, зависит от конкретного сервера. Например, если функция должна вернуть список каких-либо объектов, она может вернуть S_OK, если список не пуст, и S_FALSE, если ошибок не было, но список пустой.
E_FAIL	Ошибка без указания причины.
E_UNEXPECTED	"Катастрофическая" ошибка - непредвиденная

	ситуация, из-за которой операция не может быть выполнена.
E_NOTIMPL	Функция не реализована. Если по какой-то причине разработчик сервера не считает нужным реализовывать какие-либо функции интерфейса, он пишет их так, чтобы они в любом случае возвращали это значение.
E_OUTOFMEMORY	Нехватка памяти.
E_INVALIDARG	Неверный аргумент функции
E_NOINTERFACE	Запрошен интерфейс, отсутствующий в сервере
E_POINTER	Неверный указатель
E□HANDLE	Неверный дескриптор
E_ABORT	Операция прервана
E_ACCESSDENIED	В доступе отказано

В таблице приведены в качестве примера некоторые константы типа **HRESULT**. Разработчик **DCOM-сервера** может вводить свои константы для обозначения ошибок, специфичных для данного сервера. При наименовании констант Microsoft рекомендует начинать их с "**S_**" (или вставлять в середину **"_S_"**), если подразумевается успешное завершение операции и начинать константу с **"E_"** (или вставлять в середину **"_E_"**), если кодируется ошибка.

2.2.3 Серверы в COM/DCOM

В модели COM/DCOM серверы делятся на **внутренние (in-process)**, **локальные (local)** и **удалённые (remote). Внутренние серверы** - это динамически компонуемые библиотеки **(Dynamically Linked Library, DLL)**, которые при необходимости отображаются в адресное пространство клиентского процесса. **Локальные серверы** - это исполняемые файлы, которые запускаются независимо от клиента. **Удалённые серверы** - это исполняемые файлы, работающие на другом компьютере. Один и тот же исполняемый файл может быть и локальным, и удалённым - это зависит от того, какой клиент его использует. Более того, **COM/DCOM-серверы** могут обслуживать несколько клиентов одновременно, поэтому один и тот же сервер может быть локальным и удалённым одновременно. В дальнейшем мы будем использовать термин **"внутренний сервер"** для

обозначения серверов, размещённых в **DLL**, и **"внешний сервер"** - для размещённых в исполняемом файле.

Местоположение сервера является прозрачным для клиента. В принципе, клиент при соединении может указать, сервер какого типа (внутренний, локальный или удалённый) ему нужен, но обычно эта возможность не используется. Дальнейшая работа с сервером не зависит от его типа, клиент может даже не знать, с сервером какого типа он работает. Если клиент работал с внутренним сервером, а затем вышла следующая версия сервера, совместимая со старой, но размещённая в исполняемом файле, клиент сможет с ней работать без модификации.

2.2.4 Коклассы

COM/DCOM - это объектная модель, вся работа строится на объектах. Каждый **COM-объект** имеет тип, называемый **коклассом** (**CoClass**; префикс **"Co"** указывает на принадлежность к технологии **COM**).

COM-сервер может создавать свой **COM-объект** для обработки вызовов от каждого клиента или использовать один **COM-объект** для обработки вызовов всех клиентов. Кроме того, один сервер может содержать несколько разных коклассов.

Именно **кокласс**, а не сервер является центральным объектом для **COM-клиента**. Если клиент работает с разными **коклассами**, то он не может узнать, реализуются ли они одним сервером или разными. Каждый **кокласс** имеет своё **имя** и **GUID**, называемый в данном случае **CLSID**. Имя - это любая последовательность символов, называемая **ProgID** (Обратите внимание, что **ProgID** не является **GUID**).

Обычно **ProgID** состоит из имени разработчика, имени кокласса и номера версии, разделёнными точками (например, **"SomeCompany.SomeCoclass.1"**). Имя **кокласса** отражает его функциональное предназначение. При регистрации кокласса в системный реестр заносится и **ProgID**, и **CLSID**, поэтому, зная один из этих идентификаторов, можно узнать и второй. Для этого можно использовать системные функции **CLSIDFromProgID** и **ProgIDFromCLSID**.

Кроме **ProgID**, существует ещё и **VersionIndependentProgID** - независимый от версии **ProgID**. Обычно он отличается от **ProgID** только отсутствием номера версии в конце строки. Если на компьютере установлено несколько версий одного кокласса, каждая из них должна иметь свой **CLSID** и **ProgID**, но **VersionIndependentProgID** будет у них общим. Если в функцию **CLSIDFromProgID** передать в качестве параметра **VersionIndependentProgID**, она вернёт **CLSID** самой последней версии кокласса.

2.2.5 Интерфейсы

Другим не менее важным понятием технологии **COM/DCOM** является интерфейс. **Интерфейс** - это список функций. С точки зрения двоичного кода интерфейс представляет собой таблицу, содержащую указатели на функции. Количество элементов в этой таблице соответствует количеству функций в интерфейсе. Сервер реализует функции, создаёт таблицы с указателями на них, а указатель на эту таблицу может быть получен клиентом. **Именно через эти указатели клиент получает доступ к функциям сервера**. Количество интерфейсов, которые может экспортировать один **кокласс**, не ограничивается.

Каждый **интерфейс** имеет **имя** и уникальный **GUID**, называемый в данном случае **IID**. Имя интерфейса должно быть идентификатором, корректным с точки зрения языка, на котором разрабатывается клиент или сервер. Так как различные языки программирования имеют разные требования к идентификаторам, в имени интерфейса рекомендуется использовать только английские буквы и символ подчёркивания "_". Кроме того, многие языки нечувствительны к регистру символов, поэтому нежелательно давать разным интерфейсам имена, отличающиеся лишь регистром символов. И, наконец, названия интерфейсов принято начинать с буквы **"I".**

Однажды созданный и опубликованный интерфейс не должен меняться, потому что иначе клиент, получивший указатель на данный интерфейс, может попытаться вызвать функцию, которой он не содержит или передать ей не те параметры. Если интерфейс устарел, то вместо его модификации нужно создать новый интерфейс, оставив старый без изменения.

При создании интерфейсов их можно наследовать от уже имеющихся. Интерфейс-наследник может добавлять новые функции, но перекрытие старых функций не имеет смысла, потому что интерфейс не содержит реализации функций. Сам по себе бинарный код, в отличие от обычного ООП, в модели **COM/DCOM** никогда не наследуется, **наследуются только интерфейсы, т. е. декларация функций**.

При создании **COM-сервера** разработчик должен прежде всего решить, может ли он обойтись разработанными ранее (им же или кем-то другим) интерфейсами или же ему нужно разработать свои интерфейсы. В последнем случае он планирует, какие функции должен содержать каждый из новых интерфейсов, придумывает им имена и присваивает каждому уникальный **IID**. Затем начинается этап реализации функций этих интерфейсов.

2.2.6 Фабрика классов

Реализация **кокласса** в сервере может быть различной в зависимости от выбранного языка программирования, назначения кокласса, реализуемых им интерфейсов и т. д. Однако система должна иметь возможность создать экземпляр любого кокласса, зарегистрированного в ней. Поэтому сервер должен предоставить системе механизм создания **COM-объекта**, не зависящий от особенностей его реализации. Таким механизмом является **фабрика класса** (называемая также **объектом класса**).

Фабрика класса — это специфический **COM-объект**, реализующий интерфейс **IClassFactory**.

3 Использование технологии COM в Delphi

3.1 Функции и процедуры для работы GUID

Для хранения **GUID** в System.pas определён специальный тип:

```
TGUID = packed record
   D1: LongWord;
   D2: Word;
   D3: Word;
   D4: array[0..7] of Byte;
 end;
```

Для сравнения двух **GUID**-ов в системе определена функция **IsEqualGUID**. Хотя **GUID** очень интенсивно используется в COM, он не является составной частью этой технологии. **GUID** используется везде, где необходим уникальный идентификатор. Например, тип **UniqueIdentifier** в **SQL Server** - это **GUID**.

Сгенерировать новый **GUID** можно с помощью стандартной функции Windows **CoCreateGuid**. Начиная с версии **Delphi 6**, в **модуле SysUtils** наряду с **CoCreateGUID** описана функция **CreateGUID**, которую можно использовать как в Windows, так и в Linux. Внутри для Windows она просто вызывает **CoCreateGuid**. Для генерации нового GUID в редакторе кода **Delphi** нужно нажать сочетание клавиш **"Ctrl+Shift+G"**.

Вот еще ряд функций наиболее часто используемые при работе с GUID:

ClassIDToProgID Возвращает идентификатор ProgID для заданного класса.

CreateClassID Генерирует новый уникальный идентификатор GUIID.

GUIDToString Возвращает строковое имя, соответствующее идентификатору класса GUID.

OleCheck Вызывает исключение EOleSysError.

OleError Вызывает исключение EOleSysError.

ProgIDToClassID Возвращает идентификатор класса CLSID, который соответствует указанному идентификатору ProgID.

StringToGUID Преобразовывает имя идентификатора GUID в значение CLSID формата TGUID.

Рассмотрим использование функций для работы с GUID на примере.

Ниже приведен код модуля и скриншот выполнения программы (рисунок 3.1)

```
unit Test_GUID;
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ComObj, StdCtrls,ActiveX;

procedure TForm1.Button1Click(Sender: TObject);
var
G,Gl: TGUID;
W: PWideChar;
I: Integer;
begin
OleCheck(CLSIDFromProgID('InternetExplorer.Application', G));
OleCheck(StringFromCLSID(G, W));
Memo1.Lines.Add(W);
CoTaskMemFree(W);
W := CoTaskMemAlloc(100 * SizeOf(WideChar));
StringFromGUID2(G, W, 100);
Memo1.Lines.Add(W);
CoTaskMemFree(W);
OleCheck(ProgIDFromCLSID(G,W));
Memo1.Lines.Add(W);
CoTaskMemFree(W);
OleCheck(IIDFromString('{0002DF01-0000-0000-C000-111111111111}', G));
OleCheck(StringFromIID(G,W));
Memo1.Lines.Add(W);
CoTaskMemFree(W);
for I := 0 to 9 do begin
OleCheck(CoCreateGuid(G));
OleCheck(StringFromIID(G, W));
Memo1.Lines.Add(W);
CoTaskMemFree(W);
end;
OleCheck(IIDFromString('{0002DF01-0000-0000-C000-111111111111}', G));
if IsEqualGUID(G, Gl) then
Memo1.Lines.Add('EQUAL')
else
```

```
Memo1.Lines.Add('NOT EQUAL');
if IsEqualGUID(G, G) then
Memo1.Lines.Add('EQUAL')
else
Memo1.Lines.Add('NOT EQUAL')
end;
```

Результаты выполнения программы

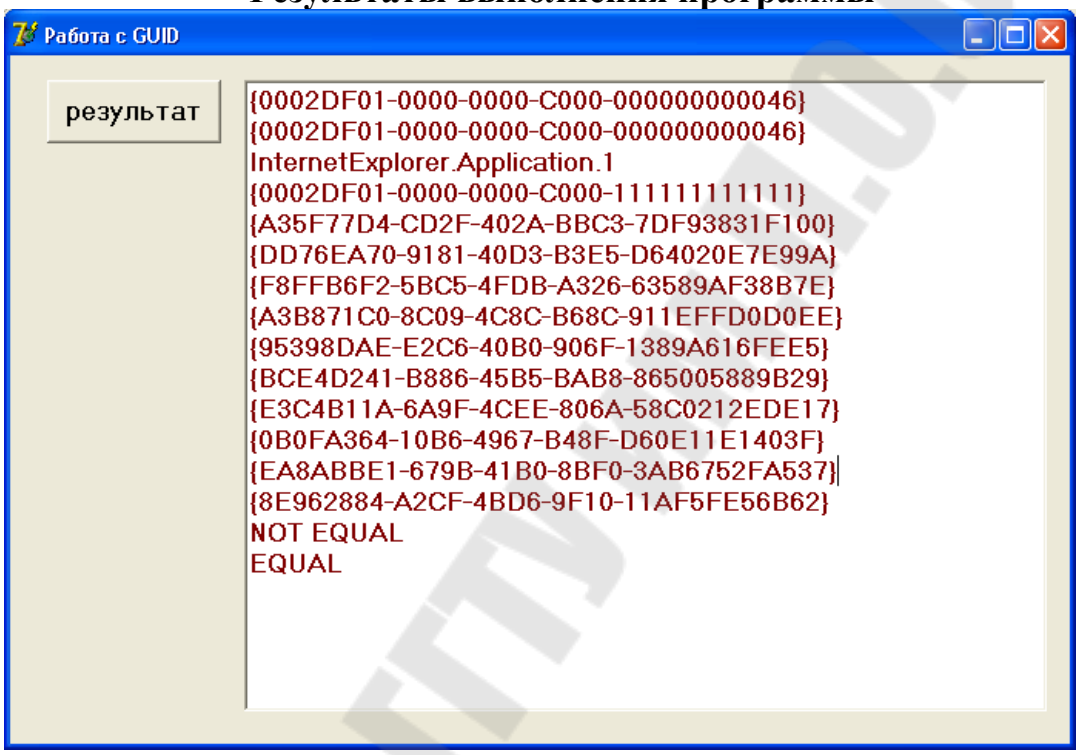

Рисунок 3.1

3.2 Поддержка интерфейсов в Delphi

Для поддержки интерфейсов в **Delphi** введено ключевое слово **interface**, с помощью которого можно объявить интерфейсный тип:

```
type <InterfaceName> = interface[()]
[<IID>]
  { <Method> | <Property> }
end;
```

IID интерфейса записывается в виде обрамлённой апострофами строки, заключённой в квадратные скобки, например:

['{00000002-0000-0000-C000-000000000046}'].

Указание **IID** при объявлении интерфейсного типа необязательно, но интерфейс без **IID** не совместим с **COM/DCOM** и имеет ограниченное применение внутри программы.

Объявление интерфейса похоже на объявление класса с некоторыми ограничениями:

- **Интерфейс не может содержать полей, свойства должны читаться и записываться только с помощью методов. Нельзя задавать значение по умолчанию.**
- **Все элементы интерфейса имеют видимость public, недопустимо явное указание области видимости.**
- **Интерфейс не имеет конструкторов и деструкторов.**
- **Методы не могут быть объявлены как virtual, dynamic, abstract, override, message.**

Переменные интерфейсного типа, так же, как и переменные классовых типов, являются указателями. Разыменование этих указателей компилятор при необходимости выполняет неявно, поэтому использовать символ **"^"** нужды нет.

Если не указан предок интерфейса, предком является базовый интерфейс **IUnknown** (подробно он будет описан в следующем разделе). В отличие от класса, **интерфейс никогда не содержит реализации своих методов**, только их описания (в этом смысле методы интерфейса схожи с абстрактными методами класса). **Реализация интерфейса в Delphi всегда осуществляется в классе**. Один класс может реализовывать сразу несколько интерфейсов. Реализуемые интерфейсы перечисляются в строке объявления после указания класса-предка. Обязательно явное указание предка, даже если это TObject.

```
TMyClass=class(TSomeClass, ISomeInterface1, ISomeInterface2…)
```

Как и при наследовании от классов с абстрактными методами, класс, реализующий интерфейсы, должен содержать **ВСЕ** их методы, точно соответствующие по именам и спискам параметров. Если класс реализует два интерфейса, один из которых является наследником другого, их общие методы объявляются и реализуются классом один раз, и эта единственная реализация становится общей для обоих интерфейсов. Но возможны также ситуации, когда два независимых интерфейса содержат одноимённые методы. В этом случае каждый из одноимённых методов должен иметь отдельную реализацию, а чтобы избежать конфликта имён, используются **псевдонимы**. Пусть интерфейсы **ISomeInterface1** и **ISomeInterface2** содержат одноимённые методы **Test**. Следующий пример показывает использование псевдонимов для разрешения конфликта имён в таком случае:

```
TMyClass=class(TSomeClass, ISomeInterface1, ISomeInterface2)
private
        procedure ISomeInterface1.Test=Test1;
        procedure ISomeInterface2.Test=Test2;
        procedure Test1;
        procedure Test2;
end;
```

Еще одно ключевое слово, связанное с реализацией интерфейса - это директива **implements**.

```
type <ClassName> = class(<ParentClass> {,<Interface>})
property <PropertyName> : <Class> | <Interface> read <Field> | <Method>
          write <Field> | <Method> implements <Interface> {,<Interface>}
end;
```

Директива **implements** позволяет делегировать реализацию методов интерфейса другому классу. Такой подход даёт возможность разбить реализацию сложного класса на несколько простых, что упрощает программирование и повышает модульность программы. Делегирование интерфейсов имеет ряд особенностей, которые подробно будут описаны в другом разделе.

Получить интерфейс, реализуемый объектом, можно с помощью метода:

```
TObject.GetInterface(const IID: TGUID; out Obj): Boolean;
```

В качестве параметра Obj должна передаваться переменная типа интерфейс. Если объект поддерживает интерфейс с заданным IID, функция вернет True, а в параметр Obj будет записан указатель на требуемый интерфейс, в противном случае будет возвращено False, а Obj получит значение nil.

Поскольку **IID** интерфейса описывается как часть объявления, компилятор знает о том, как получить его. Везде, где необходимы параметры типа **TIID** или **TGUID**, можно передать имя интерфейса.

Например,

```
if MyClass.GetInterface(ISomeInterface1, Obj) then…
```

В Delphi предусмотрен наследование интерфейсов

```
Type
ISomeInterface1=interface
        // определение ISomeInterface1
        end;
ISomeInterface2=interface(ISomeInterface1)
        // определение ISomeInterface2
        end;
        TMyClass=class(TSomeClass, ISomeInterface2)
```

Для реализации **ISomeInterface2 TMyClass** должен реализовать методы **ISomeInterface1** и **ISomeInterface2**. Но **MyClass.GetInterface(ISomeInterface1, Obj)** вернет **False**. Несмотря на то, что начало таблицы **ISomeInterface2** совпадает с таблицей **ISomeInterface1** и указатель на **ISomeInterface2** может использоваться как указатель на **ISomeInterface1**, функции **GetInterface** это неизвестно. **Класс "знает" об интерфейсах, о поддержке которых он или его предок объявил явно.**

Интерфейсы можно также рассматривать как частичную замену множественному наследованию классов, отсутствующему в Delphi. Множественное наследование считается мощным, но небезопасным инструментом, способным привести к трудно обнаруживаемым ошибкам. При использовании интерфейсов эти ошибки не возникают, т.к. они связаны с наследованием реализации, а в данном случае наследуются только объявления. И хотя интерфейсы не могут полностью заменить множественное наследование классов, они являются разумным компромиссом между мощностью и безопасностью.

Пример использования интерфейсов

```
unit Test_Interface;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  Itest1=interface
```

```
function Plosch(a,b:real):real;
function Perimetr(a,b:real):real;
end;

Itest2=interface
function Plosh(r:real):real;
function Dlina(r:real):real;
end;

TTest=class(TInterfacedObject,IInterface,Itest1,Itest2)
function Plosch(a,b:real):real;
function Perimetr(a,b:real):real;
function Plosh(r:real):real;
function Dlina(r:real):real;
end;


TForm1 = class(TForm)
  GroupBox1: TGroupBox;
  GroupBox2: TGroupBox;
  Edit1: TEdit;
  Edit2: TEdit;
  Edit3: TEdit;
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  Label5: TLabel;
  Label6: TLabel;
  Label7: TLabel;
  Label8: TLabel;
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
end;


var
 Form1: TForm1;
```

```
implementation

{$R *.dfm}

function TTest.Plosch(a,b:real):real;
begin
result:=a*b;
end;

function TTest.Perimetr(a,b:real):real;
begin
result:=2*(a+b);
end;

function TTest.Plosh(r:real):real;
begin
result:=pi*sqr(r);
end;

function TTest.Dlina(r:real):real;
begin
result:=2*pi*r;
end;


procedure TForm1.Button1Click(Sender: TObject);
var a,b,r,S,P,Skr,Dl:real;
Pr:ITest1;
Kr:ITest2;
begin
a:=strtofloat(edit1.text);
b:=strtofloat(edit2.text);
r:=strtofloat(edit3.text);
Kr:=TTest.Create;
Skr:=Kr.Plosh(r);
Dl:=Kr.Dlina(r);
Pr:=TTest.Create;
S:=Pr.Plosch(a,b);
P:=Pr.Perimetr(a,b);
Label5.Caption:=FloatToStr(S);
Label6.Caption:=FloatToStr(P);
Label7.Caption:=FloatToStr(Skr);
```

```
Label8.Caption:=FloatToStr(Dl);
end;

end.
```

На рисунке 3.2 представлены результаты работы программы

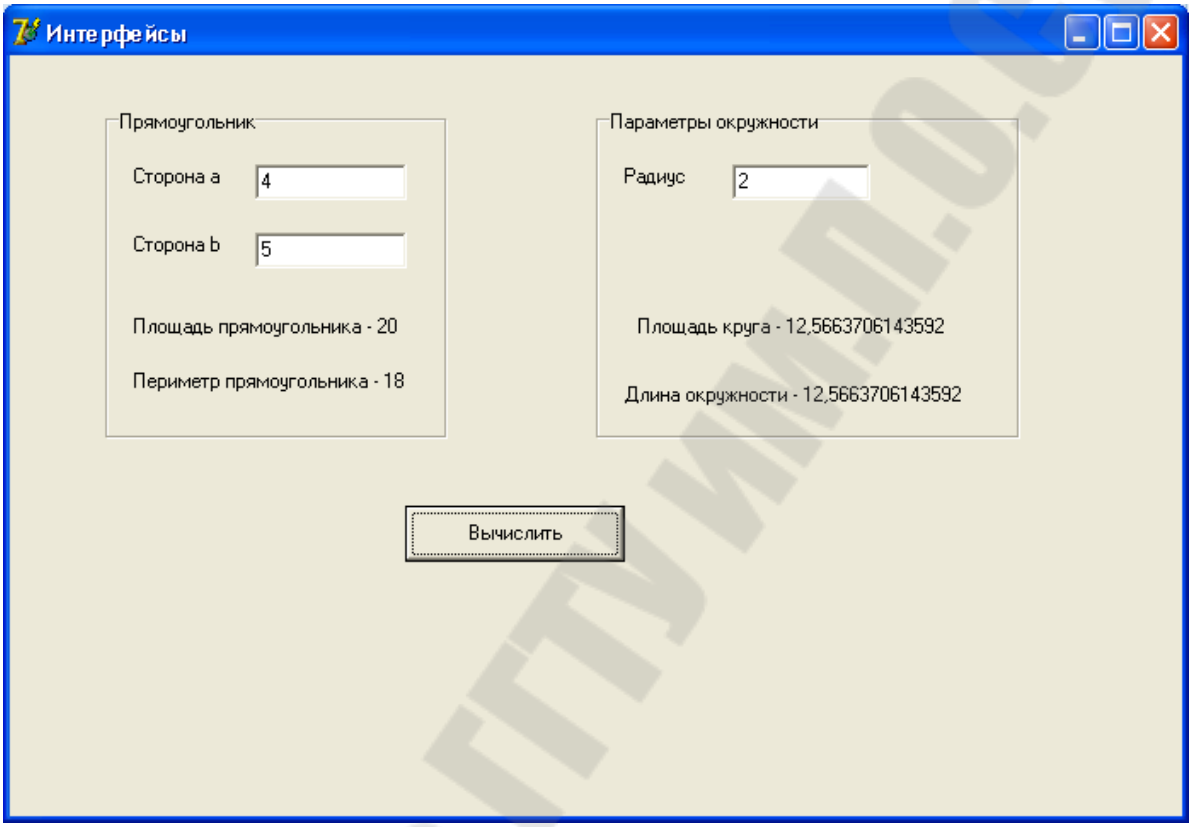

Рисунок 3.2

3.3 Интерфейс IUnknown

Интерфейс **IUnknown** является базовым интерфейсом для **COM/DCOM**. Все интерфейсы должны быть его наследниками (прямыми или косвенными). Никакие исключения из этого правила не допускаются.

Интерфейс **IUnknown** определяется в модуле **System.pas** следующим образом:

```
Type
IUnknown = interface
["{00000000-0000-0000-C000-000000000046}"]
function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
function _AddRef: Integer; stdcall;
```

```
function _Release: Integer; stdcall;
end;
```

Функция **QueryInterface** служит для получения указателя на интерфейс **COM-объекта**. Так как каждый из интерфейсов является наследником **IUnknown**, в каждом из них есть функция **QueryInterface**. Таким образом, имея указатель на один из интерфейсов **COM-объекта**, можно получить указатель на любой другой его интерфейс.

Параметр **IID** специфицирует **IID** требуемого интерфейса, в параметр **Obj** будет записан указатель на требуемый интерфейс. Если кокласс не содержит интерфейс с заданным **IID**, функция должна вернуть **E_NOINTERFACE**. Кроме того, в **Obj** в этом случае должно быть значение **nil**. **(Вообще, это общее правило для COM/DCOM: если функция завершается с ошибкой, она должна все свои выходные параметры, являющиеся указателями, установить в nil.)**

Рассмотрим ситуацию, когда кокласс содержит некоторый интерфейс **IInterface2**, унаследованный от **IInterface1**.

```
Type
IInterface1=interface
        // определение IInterface1
        end;
IInterface2=interface(IInterface1)
        // определение IInterface2
        end;
TMyClass=class(TSomeClass, ISomeInterface2)
```

Они имеют идентификаторы **IID2** и **IID1** соответственно. Данный **кокласс** может использоваться клиентом, который ничего не знает про **IInterface2**, и поэтому попытается запросить **IInterface1**. Спецификация COM/DCOM требует, чтобы функция **QueryInterface** корректно отрабатывала этот вызов и возвращала **IInterface1** (те классы, которые реализуют интерфейсы не для COM/DCOM, могут не следовать этому правилу). Если цепочка наследования интерфейсов более длинная, **функция QueryInterface должна возвращать указатель на любой из интерфейсов** этой цепочки. В частности, это означает, что через **QueryInterface** можно получить указатель непосредственно на интерфейс **IUnknown**, так как он стоит в начале любой цепочки наследования.

Функции **_AddRef** и **_Release** служат для управления временем жизни **COM-объекта**, экспортирующего интерфейсы. **_AddRef** увеличивает число ссылок на данный **COM-объект** на единицу, **_Release** уменьшает его. Если при очередном вызове **_Release** число ссылок

оказалось равным нулю, **COM-объект** должен уничтожиться и освободить все ресурсы.

В документации Microsoft по COM/DCOM не используется символ подчеркивания перед функциями **_AddRef** и **_Release**. С точки зрения COM/DCOM это не имеет значения, принципиальным является порядок следования, количество и типы параметров и модель вызова функции. Для Delphi эти методы являются особенными. Компилятор автоматически вызывает эти методы для того, чтобы обеспечить правильный подсчёт ссылок. Каждый раз, когда переменная интерфейсного типа меняет своё значение, для её старого значения, если оно не равно **nil**, вызывается **_Release**, а для нового, если оно не равно **nil** - **_AddRef**. Когда переменная выходит из области видимости, для неё также вызывается **_Release**, если она не равна **nil**. Компилятор также устанавливает любым интерфейсным переменным начальное значение **nil**, чтобы первое присвоение не приводило к вызову **_Release** по **"мусорному" адресу**. **Программист не должен вызывать функции _AddRef и _Release напрямую.**

Спецификация COM/DCOM требует, чтобы функции **_AddRef** и **_Release** возвращали любое ненулевое значение, если счётчик ссылок объекта не равен нулю, и нулевое - если равен. На практике они обычно возвращают значение счётчика. Тем не менее, COM/DCOM разрешает клиентам использовать возвращаемые этими функциями значения только в отладочных целях. В окончательной версии программы эти значения должны игнорироваться. Программисту на Delphi очень легко следовать данному правилу: так как **_AddRef и _Release вызываются компилятором неявно**, получить доступ к возвращаемым значениям просто невозможно.

Рассмотрим пример:

```
type
  IInterface1=interface  // прямой наследник IUnknown
["{FD054108-7B86-4D1A-BDE3-887465E00099}"]
    function Func1:HRESULT;
   end;

   IInterface2=interface(IInterface1)
["{193464E5-DD2B-4A1B-B469-26B6A9F1F766}"]
  function Func2: HRESULT;
   end;

   TTest=class(TObject,IUnknown,IInterface2)
   private
     FRefCount: Integer; // счетчик ссылок
```

```
        { IUnknown }
        function QueryInterface(const IID: TGUID; out Obj): HRESULT;
stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
        { IInterface1 }
        function Func1: HRESULT;
        { IInterface1 }
        function Func2: HRESULT;
      public
       constructor Create;
      end;

{ TTest реализация }
constructor TTest.Create;
 begin
  inherited Create;
  FRefCount:=0
  { Первоначально на вновь созданный объект нет ссылок, поэтому
    счётчик ссылок должен быть инициализирован нулём }
 end;

function TTest.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
{используем автоматическую поддержку интерфейсов}
   if GetInterface(IID, Obj) then Result := S_OK
   else Result := E_NOINTERFACE;
end;

function TTest._AddRef: Integer;
begin
  Inc(FRefCount); // увеличение числа ссылок
  Result:=FRefCount;
end;

function TTest._Release: Integer;
begin
  Dec(FRefCount); // уменьшение числа ссылок
  Result := FRefCount;
  if Result = 0 then // удаление объекта
    Destroy;
end;
```

Интерфейсы "совместимы" с объектами, их поддерживающими. В кавычках, потому что внутренний механизм отличается от совместимости объектов.

```
var
  I1:IInterface1;
  I2:IInterface2;
begin
  I2:=TTest.Create;
{Получение интерфейса на этапе компиляции}
I1:=TTest.Create;
{А эта строчка выдаст ошибку компиляции
   Incompatible types; класс не объявил о поддержке IInterface1}
```

Используя один интерфейс, можно получить другой с помощью оператора **as**. Оператор **as**, применённый к интерфейсному объекту, означает вызов **QueryInterface**. Под интерфейсным объектом подразумевается либо другой интерфейс, либо объект, реализующий интерфейс. Например,

```
I1:=TTest.Create as Interface1;
```

Это означает получение интерфейса **IUnknown** на этапе компиляции (из чего следует, что **IUnknown** должен присутствовать в объявлении класса или его предка). Затем вызов метода **QueryInterface**. Если **QueryInterface** не вернет **S_OK**, возникнет исключительная ситуация **"Interface not supported".** Т. е. произойдет ошибка времени исполнения.

Вернёмся к приведённому выше примеру реализации **TTest**. Удалим из него все упоминания об интерфейсе **IInterface2**, оставив только реализацию **IUnknown**. Теперь породим от него наследника, реализующего **IInterface2**:

```
class TTest2=class(TTest,IInterface2)
```

В классе **TTest2** мы теперь должны будем реализовать только методы **Func1** и **Func2**, а реализацию методов **IUnknown** будет унаследована от **TTest**. Функция **GetInterface**, использованная в **QueryInterface**, определяет набор интерфейсов, реализуемых классом, динамически, поэтому сможет "почувствовать", что объект реализует ещё и **IInterface2**. Легко видеть, что **TTest** теперь можно использовать как базовый класс для любых других классов, реализующих интерфейсы.

Таким образом, **IUnknown** достаточно реализовать один раз в **TTest**, а потом эту реализацию можно использовать как базу для всех остальных классов, реализующих любые интерфейсы.

На самом деле нет необходимости самостоятельно создавать класс, реализующий **IUnknown**, чтобы потом порождать от него другие классы - в модуле **System** уже есть готовый такой класс, который называется **TInterfacedObject**. От нашего примитивного **TTest TInterfacedObject** отличается более ответственным подходом к подсчёту ссылок на этапе создания класса. Пусть в нашем классе **TTest2**, порождённом от **TTest**, конструктор выглядит следующим образом:

```
constructor TTest2.Create;
 var Interface2:IInterface2;
  begin
   inherited Create;
   Interface2:=Self;
   // Далее вызываются методы Interface2
  end;
```

После выполнения унаследованного конструктора счётчик ссылок будет равным нулю. Получение интерфейса **IInterface2** увеличит его на единицу. После завершения конструктора при финализации переменной **Interface2** автоматически будет вызван метод **_Release**, который уменьшит счётчик ссылок до нуля и уничтожит объект. Таким образом, экземпляр наследника **TTest** невозможно будет создать, если этот наследник в своём конструкторе использует ссылки на свои же интерфейсы.

Чтобы избежать этой проблемы, в конструкторе класса **TInterfacedObject** (точнее - в его методе **NewInstance**, который вызывается из конструктора) счётчик ссылок инициализируется не нулём, а единицей, а потом, в методе **AfterConstruction**, который вызывается после полного выполнения конструктора, счётчик уменьшается на эту лишнюю единицу. Эти дополнительные манипуляции со счётчиком обеспечивают безопасность использования своих интерфейсов в конструкторе наследника.

TInterfacedObject также следит за тем, чтобы объект освобождался только при уменьшении счётчика ссылок до нуля, т.е. за счёт неявных вызовов _Release. Если попытаться освободить такой объект явным вызовом его деструктора, метод **BeforeDestruction** инициирует ошибку времени выполнения.

Ещё одним отличием **TInterfacedObject** от **TTest** является использование синхронизации при работе со счётчиком. Таким образом,

реализация **IUnknown** в **TInterfacedObject** обеспечивает корректный подсчёт ссылок при использовании объекта несколькими нитями.

TInterfacedObject обеспечивает только реализацию методов **IUnknown**. Этого достаточно при использовании интерфейсов для внутренних нужд программы, но базовая функциональность COM-объекта должна быть существенно шире. Поэтому для создания **COM-объектов** классы обычно наследуются не от **TInterfacedObject,** а от **TComObject,** который будет рассмотрен позже.

Примечание: Начиная с версии 6, базовым интерфейсом Delphi является IInterface, который полностью соответствует IUnknown. Рекомендуется использовать IInterface для платформонезависимых приложений и IUnknown для приложений под Windows.

3.3 Использование сервера клиентом

В этом разделе мы рассмотрим действия, которые выполняет клиент для работы с сервером: **создание COM-объекта, получение указателей на его интерфейсы, вызов функций и освобождение COM-объекта**.

Интерфейсы, указатели на которые получает клиент, могут быть реализованы в рамках другого процесса, в том числе и в рамках процесса, выполняемого на другом компьютере. В данном разделе мы не будем касаться вопроса о том, как клиент может получать указатели на объекты из чужого адресного пространства, отметим только, что эта проблема решается системой прозрачно для клиента. Более подробно этот вопрос будет рассмотрен в разделе **"Маршалинг".** Также пока считаем, что клиент обращается к **COM-объектам** только из главной нити.

Начинать работу с COM клиент должен с вызова функции **CoInitialize**, единственный параметр которой зарезервирован для дальнейшего использования и должен быть равен **nil**. В разделе **"Нитевые модели"** мы подробнее рассмотрим, зачем нужна эта функция, а пока отметим только, что, если её не вызвать, последующие вызовы других функций **COM** будут завершаться с ошибкой **CO_E_NOTINITIALIZED**. Завершать работу с **COM** должен вызов **CoUninitialize**.

Как уже отмечалось выше, для COM-клиента базовым понятием является не сервер, а кокласс. Клиент создаёт экземпляр кокласса (COM-объект) и работает с ним, нисколько не заботясь о том, какой именно сервер реализует данный кокласс. COM-объект создаётся с помощью функции **CoCreateInstance**, имеющей следующие параметры:

function CoCreateInstance(const clsid: TCLSID; unkOuter: IUnknown;
dwClsContext: Longint; const iid: TIID; var pv): HResult; stdcall;

Параметр **clsid** задаёт **CLSID** кокласса, экземпляр которого нужен клиенту. Каждый кокласс, чтобы стать доступным клиентам, **должен быть зарегистрирован соответствующим образом в реестре**. **В частности, в реестре для каждого кокласса записывается имя файла (исполняемого или DLL), в котором этот кокласс реализован.** При вызове **CoCreateInstance** система находит запись в реестре, соответствующую данному **CLSID**, и загружает нужный сервер, если он не был загружен ранее. Далее управление передаётся специальному объекту сервера, называемому **фабрикой класса**, которая и создаёт объект. Внутреннее устройство фабрики класса будет рассмотрено в следующем разделе.

Строго говоря, в результате вызова **CoCreateInstance** не всегда создаётся новый **COM-объект**. Фабрика класса может быть написана так, что все клиенты будут использовать один и тот же **COM-объект**.

Параметр **unkOuter** используется только при агрегации, которая будет рассмотрена в разделе **"Агрегация и контейнирование".** Для обычных клиентов этот параметр должен иметь значение **nil**.

Параметр **dwClsContext** позволяет клиенту управлять типом сервера. Определены константы: **CLSCTX_INPROC_SERVER** (внутренний сервер), **CLSCTX_LOCAL_SERVER** (локальный сервер), **CLSCTX_REMOTE_SERVER** (удалённый сервер) и **CLSCTX_INPROC_HANDLER** (суррогатный процесс, загружающий внутренний сервер в своё адресное пространство). Кроме того, определены константы **CLSCTX_SERVER** (объединяет первые три константы) и **CLSCTX_ALL** (объединяет все четыре константы). Установкой нужного значения параметра **dwClsContext** клиент указывает, какие типы серверов для него приемлемы.

Суррогатный процесс - это специальная программа, позволяющая получать доступ к внутренним серверам, работающим на другом компьютере. В состав Windows NT 4.0 начиная с Service Pack 2 входит программа **dllhost.exe**, являющаяся простейшим суррогатным процессом. Она загружает в своё адресное пространство любой внутренний COM-сервер, и сама становится внешним COM-сервером, экспортирующим те же самые интерфейсы. При вызове функций этих интерфейсов удалённым клиентом **dllhost** переадресовывает их загруженному внутреннему серверу. Другим популярным суррогатным процессом является **Microsoft Transaction Server,** который не только осуществляет загрузку внутренних серверов в своё адресное пространство, но и предоставляет множество дополнительных функций.

Параметр **dwClsContext** функции **CoCreateInstance** - это единственное место, где клиент "чувствует" тип сервера. Все последующие операции с **COM-объектом** проходят одинаково независимо от типа сервера.

Выше мы видели, что, зная указатель на любой интерфейс COM-объекта, клиент с помощью функции **QueryInterface** может получить указатель на любой другой интерфейс. Но мы не затронули вопроса о том, как получить самый первый указатель на интерфейс. Он получается с помощью функции **CoCreateInstance**. Параметр **iid** задаёт **IID** желаемого интерфейса, а **pv** - переменную, в которую должен быть записан указатель на него.

Для интерфейса, полученного через **CoCreateInstance**, фабрика класса вызывает функцию **_AddRef**, поэтому он, так же, как и остальные интерфейсы, должен быть освобождён в конце работы с помощью функции **_Release**.

После получения указателя на интерфейс клиент может вызывать его функции и получать указатели на другие интерфейсы. Работа с COM-объектом заканчивается тогда, когда для всех интерфейсов вызвана функция **_Release**. Если нет других клиентов, использующих данный COM-объект, он выгружается из памяти.

В среде Delphi правильный подсчет ссылок обеспечивает компилятор. Функция **_Release** вызывается автоматически, когда переменная интерфейса выходит за пределы области видимости или принимает значение **nil**.

Система не накладывает ограничений на число серверов, с которым клиент может работать одновременно. Клиент может последовательно разместить несколько вызовов функции **CoCreateInstance** для создания различных COM-объектов и работать одновременно со всеми их интерфейсами.

В среде Delphi создание **COM-объекта** немного проще. Первое, что нужно сделать - это подключить модуль **ComObj.pas.** В разделе **initialization** этого модуля изменяется стандартная процедура инициализации. Это приведет к тому, что **Application.Initialize** автоматически вызовет **CoInitialize**. А в разделе **finalization,** если вызов CoInitialize прошел успешно, автоматически вызывается **CoUninitialize**. Таким образом, в главной нити приложения не нужно самостоятельно вызывать эти функции.

В этом же модуле объявлены функции, с помощью которых код будет выглядеть компактнее.

```
function CreateComObject(const ClassID: TGUID): IUnknown;
```

Функция получает в качестве параметра **CLSID** требуемого объекта и возвращает интерфейс **IUnknown**. Если объект не создан, создается исключительная ситуация. Внутри эта функция представляет собой простую оболочку **CoCreateInstance** с параметрами, которые программисты Borland сочли наиболее вероятными. В качестве параметра **dwClsContext** задается значение **CLSCTX_INPROC_SERVER** or **CLSCTX_LOCAL_SERVER** (внутренний или локальный сервер), в качестве требуемого интерфейса - интерфейс **IUnknown**. Результат функции CoCreateInstance проверяется с помощью процедуры **OleCheck**

3.4 Реализация фабрики классов

Напомним , что фабрика класса — это специфический **COM-объект**, реализующий интерфейс **IClassFactory**. Этот интерфейс определён в **AxtiveX.pas** следующим образом:

```
IClassFactory = interface(IUnknown)
['{00000001-0000-0000-C000-000000000046}']
  function CreateInstance(const unkOuter: IUnknown;
    const iid: TIID; out obj): HResult; stdcall;
  function LockServer(fLock: BOOL): HResult; stdcall;
end;
```

Функция **CreateInstance** непосредственно создаёт **COM-объект** и возвращает указатель на один из его интерфейсов. Параметры этой функции эквивалентны одноимённым параметрам функции **CoCreateInstance**. **CLSID кокласса задавать не нужно, так как фабрика класса создаётся только для одного кокласса. Если сервер содержит несколько коклассов, для каждого из них создаётся своя фабрика класса.**

Задавать контекст COM-объекта (параметр **dwClxContext** в функции **CoCreateInstance**) в фабрике класса также нет нужды. Фабрика класса работает в том же контексте, что и сам COM-объект, поэтому контекст определяется до вызова функции **CreateInstance**.

Так как фабрика класса и соответствующий ей кокласс — части одной и той же программы, внутри функции **CreateInstance COM-объект** создаётся обычными методами того языка, на котором написан сервер. Например, если сервер написан на Delphi, а кокласс реализован классом (как в примере, рассмотренном в разделе **"Интерфейс IUnknown"**), то создать новый COM-объект можно с помощью конструктора **Create**. И именно при реализации функции **CreateInstance** программист определяет, будет ли каждый раз создаваться новый **COM-объект** или же он будет

создан один раз, а всем клиентам будут возвращаться указатели на интерфейсы этого единственного COM-объекта.

Функция **LockServer** позволяет управлять процессом освобождения ресурсов сервером. Обычно сервер выгружается из памяти после того, как будут уничтожены все **COM-объекты**, созданные в нём. Функция **LockServer** с аргументом **TRUE** блокирует выгрузку сервера из памяти. Клиент может пользоваться функцией Lock, если он в дальнейшем планирует снова создавать COM-объекты. Это позволяет не тратить время на выгрузку и повторную загрузку сервера. Чтобы разблокировать сервер, нужно вызвать функцию **LockServer** с аргументом **FALSE** столько раз, сколько раз перед этим она была вызвана с аргументом **TRUE**.

Фабрика класса не регистрируется в реестре и вообще не имеет собственного **CLSID**. Реализуется она так же, как и остальные коклассы. Так же надо реализовать функции **QueryInterface**, **_AddRef** и **_Release**, потому что **IClassFactory**, как и все остальные COM-интерфейсы, является наследником **IUnknown**.

Внутренний сервер должен экспортировать функцию **DllGetClassObject**, создающую фабрику класса и возвращающую указатель на один из её интерфейсов. Прототип функции **DllGetClassObject** выглядит следующим образом:

```
function DllGetClassObject(
  const clsid: TCLSID;
  const iid: TIID;
  out pv): HResult; stdcall;
```

Через параметр **clsid** передаётся **CLSID кокласса**, для которого нужно создать фабрику класса. Параметр **iid** содержит IID интерфейса, указатель на который должен быть возвращён в параметре **pv**. Обращаем внимание, что имеется ввиду интерфейс фабрики класса, а не кокласса, потому что соответствующий **COM-объект** будет создан только после вызова **IClassFactory.CreateInstance.**

На первый взгляд может показаться странным, что при вызове **DllGetClassObject** нужно указывать **IID** интерфейса, потому что фабрика класса реализует только интерфейс **IClassFactory**. На самом деле в некоторых ситуациях реализуется не **IClassFactory**, а его наследник — **IClassFactory2**. Этот интерфейс содержит дополнительные функции, через которые клиент должен подтвердить, что у него есть лицензия на использование данного сервера. Если производитель сервера хочет защитить свой продукт лицензией, он реализует в фабрике класса не **IClassFactory**, а **IClassFactory2**. Кроме того, нельзя исключить, что в

будущем возможности фабрики класса будут расширены, и потребуются новые интерфейсы.

Фабрика класса, созданная в результате вызова **DllGetClassObject**, уничтожается обычным для COM-объектов способом: для этого нужно освободить все её интерфейсы. В принципе, нет необходимости создавать новую фабрику класса каждый раз при вызове **DllGetClassObject** — можно создать её один раз и возвращать ссылку на интерфейсы одного и того же экземпляра.

Внешний сервер предоставляет системе указатель на свою фабрику класса иначе. Сначала он создаёт её, а затем сообщает системе, что она создана, с помощью функции **CoRegisterClassObject**. Через эту функцию система получает указатель **CLSID** кокласса, для которого создана фабрика класса, а так же указатель на интерфейс **IUnknown** фабрики класса. Если сервер содержит несколько типов коклассов, для фабрики классов каждого из них должна быть вызвана функция **CoRegisterClassObject**. Перед уничтожением фабрики класса внешний сервер сообщает об этом системе с помощью функции **CoRevokeClassObject.**

Клиент, желающий создать **COM-объект**, должен получить указатель на интерфейс соответствующей **фабрики класса**. Для этого он вызывает функцию **CoGetClassObject**, имеющую (согласно модулю **ActiveX**) следующий прототип:

```
function CoGetClassObject(
  const clsid: TCLSID;
  dwClsContext: Longint;
  pvReserved: Pointer;
  const iid: TIID;
  out pv): HResult; stdcall;
```

Параметры функции **CoGetClassObject** аналогичны одноимённым параметрам функций **CoCreateInstance** и **CoCreateInstanceEx**.

При вызове **CoGetClassObject** система ищет в реестре кокласс с **CLSID**, определяемым параметром **clsid**. Найдя соответствующую запись, система определяет имя и тип сервера и проверяет, допускает ли параметр **dwClsContext** использование сервера данного типа. Если эта проверка прошла нормально, система проверяет, загружен ли этот сервер в адресное пространство клиента (если это внутренний сервер) или запущен ли он (если это внешний сервер). При необходимости происходит загрузка заданного сервера. Далее система получает от сервера указатель на интерфейс фабрики класса, затребованный клиентом, и возвращает его через параметр **pv**.

Третий параметр функции, являющийся нетипизированным указателем с именем **pvReserved**, на самом деле должен иметь тип **PCoServerInfo** — такой же, как и параметр **ServerInfo** функции **CoCreateInstanceEx**. С помощью этого параметра можно получать фабрику класса удалённого сервера.

Если клиент и сервер работают на разных машинах, информации о сервере в реестре клиентской машины может не быть. Поэтому в том случае, когда параметр **pvReserved** задаёт удалённую машину, запрос передаётся ей, и в её реестре происходит поиск.

Получив указатель на интерфейс фабрики класса, клиент создаёт с помощью функции **CreateInstance** столько **COM-объектов**, сколько ему нужно, а затем освобождает интерфейс. Функция **CoCreateInstance**, которую мы рассматривали ранее, выполняет именно эти действия. Она вызывает **CoGetClassObject**, создаёт с помощью полученного интерфейса требуемый COM-объект и возвращает клиенту указатель на выбранный интерфейс. После этого интерфейс **IClassFactory** освобождается, что, возможно, приводит к уничтожению фабрики класса (как мы уже говорили выше, фабрика класса может быть и сохранена сервером на тот случай, если ещё понадобится).

Если клиенту требуется только один экземпляр кокласса, лучше использовать **CoCreateInstance**, так как это избавляет программиста от рутинных операций с фабрикой класса. Но если требуется создать несколько одинаковых COM-объектов, работа с интерфейсом **IClassFactory** позволяет экономить ресурсы, так как при многократном вызове **CoCreateInstance** происходит многократное обращение к реестру и, в некоторых серверах, многократное создание и удаление фабрики класса. Кроме того, если нужен доступ к функции **LockServer** или к интерфейсу **IClassFactory2**, альтернативы прямой работе с фабрикой класса не существует.

Коклассы, экземпляры которых порождаются через фабрику класса, называются **создаваемыми.** Кроме них существуют и так называемые несоздаваемые коклассы, которые не имеют собственной фабрики класса, не регистрируются в реестре и нередко обходятся без собственного CLSID. Несоздаваемые коклассы — это вспомогательные коклассы, настолько зависимые от других коклассов, что нет смысла создавать их экземпляры отдельно. Они создаются через интерфейсы тех коклассов, от которых зависят.

При создании COM-сервера средствами Delphi писать код фабрики класса вручную обычно не нужно — её прекрасно реализуют классы **TComObjectFactory** и **TTypedComObjectFactory**.

4 Создание COM-сервера

4.1 Создание простейшего COM-сервера

Рассмотрим пример создания простейшего COM-сервера. Создавать его мы будем без использования тех средств, которые предоставляет для этого Delphi. Смысл такого самоограничения — разобраться с теми механизмами COM/DCOM, которые остаются скрыты при использовании этих самых средств Delphi.

На самом деле знаний, полученных в предыдущих лекциях, не хватает, чтобы создать полноценный COM-сервер, даже простейший. Не хватает информации о том, как помочь системе организовать вызовы методов сервера через границы процесса или сеть. Поэтому создаваемый нами сервер будет внутренним, чтобы избежать таких вызовов. Кроме того, клиент должен будет обращаться к нашему серверу только из главной нити, потому что вызовы через границы нитей мы тоже пока обеспечивать не умеем. Так что не стоит рассматривать этот пример как основу для будущих реальных серверов — он полезен только в учебных целях.

Готовый низкоуровневый сервер включает в себя четыре файла: **Test_Server.dpr, Test_Interface.pas, Test_CoClass.pas и Test_ClassFactory.pas.** Начнём с файла **Test_Interface.pas**, который содержит описание интерфейса, реализуемого нашим сервером. Секция **implementation** этого модуля пуста, а в секции **interface** объявлены следующие константы и типы:

```
unit Test_Interface;

interface

const
  // Идентификаторы сервера
  CLSID_TestServer: TGUID = '{88E1BC75-7C6B-4393-86E1-AB9CE055A84F}';
  VIProgID_TestServer = 'IPKGSTU.TestServer';
  ProgID_TestServer = VIProgID_TestServer + '.1';
  // Идентификатор интерфейса
  IID_ITestServer: TGUID = '{4E3EE84B-D631-426C-B6F6-1636ABFB9320}';

type
```

```
  ITestServer = interface(IUnknown)
    ['{4E3EE84B-D631-426C-B6F6-1636ABFB9320}']
    function setarg(a,b:real):HResult; stdcall;
    function Plosh(out Pl:real):HResult; stdcall;
    function Per(out Per:real):HResult; stdcall;
  end;

implementation

end.
```

Здесь определены константы **CLSID_TestServer, ProgID_TestServer**, которые хранят соответственно **CLSID**, **ProgID** нашего кокласса. Константа **IID_ITestServer** хранит **IID** интерфейса, который объявлен чуть ниже.

Сам **интерфейс ITestServer** содержит три функции: **setarg, Plosh** и **Per**. Функция **setarg** запоминает переданные ей в качестве параметров значения сторон прямоугольника, функция **Plosh** возвращает площадь прямоугольника, а функция **Per** возвращает периметр.

Теперь рассмотрим реализацию самого кокласса. Он реализован классом **TTestServer** в модуле **Test_CoClass**.

```
unit Test_CoClass;

interface

uses
  Windows,
  ActiveX,
  Test_Interface;

type
  TTestServer = class(TObject, IUnknown, ITestServer)
  private
    // Счётчик ссылок на объект
    FCounter: Integer;
    // Аргументы, переданные через setarg
     fa,fb:real;
    // Методы интерфейса IUnknown
    function QueryInterface(const IID: TGUID; out Obj): HRESULT;
stdcall;
    function _AddRef: Integer; stdcall;
```

```
    function _Release: Integer; stdcall;
    // Методы интерфейса Test_Interface
    function setarg(a,b:real):HResult; stdcall;
    function Plosh(out Pl:real):HResult; stdcall;
    function Per(out Per:real):HResult; stdcall;
  public
    constructor Create;
    destructor Destroy; override;
  end;

// Счётчик количества объектов
var
  ObjectsCount: Integer = 0;

implementation

constructor TTestServer.Create;
begin
  inherited Create;
  FCounter := 0;
  // Увеличиваем счётчик объектов
  Inc(ObjectsCount);
end;

destructor TTestServer.Destroy;
begin
  Dec(ObjectsCount);
  // Уменьшаем счётчик объектов
  inherited Destroy;
end;

function TTestServer.QueryInterface(const IID: TGUID; out Obj):
HResult;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else
    Result := E_NoInterface;
end;

function TTestServer._AddRef: Integer;
begin
```

```
  Inc(FCounter);
  Result := FCounter;
end;

function TTestServer._Release: Integer;
begin
  Dec(FCounter);
  Result := FCounter;
  if FCounter = 0 then
    Free;
end;

function TTestServer.setarg(a,b:real): HResult;
begin
  fa := a;
  fb := b;
  Result := S_OK;
end;

function TTestServer.Plosh(out Pl: real): HResult;
begin
  {$Q+}
  try
    Pl:= fa *fb;
    Result := S_OK;
  except
    Result := E_Fail;
  end;
end;

function TTestServer.Per(out Per:real): HResult;
begin
 {$Q+}
  try
    Per:=2*(fa+fb);
    Result := S_OK;
  except
    Result := E_Fail;
  end;
end;

end.
```

Класс реализует методы интерфейсов **IUnknown** и **ITestServer**, а также содержит **конструктор** и **деструктор** и несколько служебных полей. Реализация методов интерфейса **IUnknown** полностью соответствуют их реализации в классе **TTest**, рассмотренном в лекции, посвящённой интерфейсу **IUnknown**. Реализация методов **setarg, Plosh** и **Per** также очевидна.

Конструктор, помимо обнуления счётчика ссылок, увеличивает счётчик объектов на единицу, а деструктор уменьшает этот счётчик. **Счётчик объектов — это глобальная переменная ObjectsCount, объявленная в интерфейсной части модуля**. Этот счётчик потребуется для определения того, когда все объекты освобождены и можно выгружать сервер из памяти.

Теперь рассмотрим фабрику класса. Она реализована в модуле **Test_ClassFactory** классом **TTestClassFactory.**

```
unit Test_ClassFactory;

interface

uses
  Windows,
  ActiveX,
  Test_Interface,
  Test_CoClass;

type
  TTestClassFactory = class(TObject, IUnknown, IClassFactory)
  private
    // Счётчик ссылок
    FCounter: Integer;
    // Методы интерфейса IUnknown
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    // Методы интерфейса IClassFactory
    function CreateInstance(const unkOuter: IUnknown; const iid: TIID;
      out obj): HResult; stdcall;
    function LockServer(fLock: Bool): HResult; stdcall;
  public
    constructor Create;
    destructor Destroy; override;
```

```
  end;

// Счётчик пользовательских блокировок
var
  UserLocksCount: Integer = 0;

implementation

constructor TTestClassFactory.Create;
begin
  inherited Create;
  FCounter := 0;
  Inc(ObjectsCount);
end;

destructor TTestClassFactory.Destroy;
begin
  Dec(ObjectsCount);
  inherited Destroy;
end;

function TTestClassFactory.QueryInterface(const IID: TGUID; out Obj):
HResult;
begin
  if GetInterface(IID, Obj) then
    Result := S_OK
  else
    Result := E_NoInterface;
end;

function TTestClassFactory._AddRef: Integer;
begin
  Inc(FCounter);
  Result := FCounter;
end;
```

Реализация методов интерфейса **IUnknown** в точности совпадает с реализацией этих методов в классе **TTestServer**. Конструктор и деструктор также содержат идентичный этому классу код: так же увеличивают и уменьшают переменную **ObjectsCount**, потому что если у клиента нет ни одной ссылки на **COM-объекты** сервера, но осталась хотя бы одна ссылка на фабрику класса, такой сервер всё равно нельзя выгружать. Поэтому

переменная **ObjectsCount** считает не только сами **COM-объекты**, но и их фабрики класса.

Рассмотрим реализацию методов **IClassFactory**. Обратите внимание, что произойдёт в том случае, если клиент при создании COM-объекта попытается запросить несуществующий интерфейс. Экземпляр **TTestServer** будет создан, его счётчик ссылок изначально окажется равным нулю. При использовании оператора **as** для получения его **IUnknown** будет неявно вызван метод **QueryInterface**, который увеличит счётчик ссылок объекта на единицу. Последующий явный вызов **QureyInterface** оставит этот счётчик без изменения, т.к. метод не опознает интерфейс как допустимый и не вернёт указатель на него. В эпилог функции будет добавлен вызов метода **_Release** для освобождения неявной ссылки на **IUnknown**. В результате этого вызова счётчик ссылок объекта уменьшится до нуля, и объект уничтожит себя. Таким образом, утечки памяти не произойдёт, хотя явно мы этот объект нигде не удаляем.

Теперь вернёмся к главному модулю проекта и посмотрим на функции, которые экспортирует наша библиотека. Мы уже знаем, что она должна экспортировать функцию **DllGetClassObject.** Всего же библиотека, реализующая внутренний **COM-сервер**, должна экспортировать четыре функции:

- **DllGetClassObject.** Возвращает указатель на интерфейс фабрики класса
- **DllCanUnloadNow**. Позволяет системе определить, может ли библиотека быть выгружена без вреда для клиента.
- **DllRegisterServer.** Заносит в реестр информацию о COM-объектах сервера.
- **DllUnregisterServer.** Удаляет из реестра информацию о COM-объектах сервера.

Приведем реализацию сервера.

```
library Test_Server;

uses
  SysUtils,
  Classes,
  Registry,
  Windows,
  ActiveX,
  ComObj,
  Test_Interface in 'Test_Interface.pas',
  Test_CoClass in 'Test_CoClass.pas',
  Test_ClassFactory in 'Test_ClassFactory.pas';
```

```
{$R *.res}


function DllGetClassObject(const clsid: TCLSID; const iid: TIID; out
pv): HResult; stdcall;
var
  ClassFactory: TTestClassFactory;
begin
  // Проверяем, что COM-объект с заданным CLSID поддерживается
сервером
  if IsEqualCLSID(clsid, CLSID_TestServer) then
  begin
    // Создаём фабрику класса
    ClassFactory := TTestClassFactory.Create;
    // Возвращаем клиенту требуемый интерфейс
    Result := (ClassFactory as IUnknown).QueryInterface(IClassFactory,
pv)
  end
  else
    Result := Class_E_ClassNotAvailable;
end;

function DllCanUnloadNow: HResult; stdcall;
begin
  if (ObjectsCount <= 0) and (UserLocksCount <= 0) then
    Result := S_OK
  else
    Result := S_False;
end;

function DllRegisterServer: HResult; stdcall;
var
  Reg: TRegistry;
  ModuleName: array[0..Max_Path] of Char;
  BaseKey: string;
begin
  Reg := TRegistry.Create;
  try
    try
      Reg.RootKey := HKey_Classes_Root;
```

```
        Reg.OpenKey('CLSID', False);
        Reg.OpenKey(GUIDToString(CLSID_TestServer), True);
        BaseKey := '\' + Reg.CurrentPath;
        Reg.WriteString('', 'IPKGSTU Sample Test Server');
        Reg.OpenKey('InprocServer32', True);
        GetModuleFileName(GetModuleHandle('Test_Server.dll'),
@ModuleName[0], Max_Path + 1);
        Reg.WriteString('', ModuleName);
        Reg.OpenKey(BaseKey + '\ProgID', True);
        Reg.WriteString('', ProgID_TestServer);
        Reg.OpenKey(BaseKey + '\VersionIndependentProgID', True);
        Reg.WriteString('', VIProgID_TestServer);
        Reg.OpenKey('\' + ProgID_TestServer + '\CLSID', True);
        Reg.WriteString('', GUIDToString(CLSID_TestServer));
        Reg.OpenKey('\' + VIProgID_TestServer + '\CLSID', True);
        Reg.WriteString('', GUIDToString(CLSID_TestServer));
        Result := S_OK;
      except
        Result := E_Fail;
      end;
    finally
      Reg.Free;
    end;
  end;

  function DllUnregisterServer: HResult; stdcall;
  var
    Reg: TRegistry;
  begin
    Reg := TRegistry.Create;
    try
      try
        Reg.RootKey := HKey_Classes_Root;
        Reg.OpenKey('CLSID', False);
        Reg.DeleteKey(GUIDToString(CLSID_TestServer));
        Reg.OpenKey('\', False);
        Reg.DeleteKey(ProgID_TestServer);
        Reg.DeleteKey(VIProgID_TestServer);
        Result := S_OK;
      except
        Result := E_Fail;
      end;
```

```
  finally
    Reg.Free;
  end;
end;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

begin
end.
```

В функции **DllGetClassObject,** если клиент попытается передать **IID** интерфейса, не поддерживаемого фабрикой класса, созданная фабрика класса будет удалена при вызове **_Release** в эпилоге функции — так же, как и объект в методе **IClassFactory.CreateInstance**.

Функция **DllCanUnloadNow** должна вернуть **S_OK**, если библиотеку можно выгрузить немедленно, и **S_False**, если нельзя. Библиотеку можно выгрузить, если нет ни одного COM-объекта и ни одной фабрики класса, на которую ссылается клиент, и библиотека не заблокирована вызовом **IClassFactory.LockServer**.

Операции сравнения с нулём на всякий случай заменены операциями "меньше или равно" — чтобы сервер мог выгрузиться, если счётчики в результате некорректных действий клиента стали меньше нуля.

> **<u>Примечание:</u> В нашей реализации счётчик числа объектов защищён от того, чтобы стать меньше нуля — клиент не может удалить объектов больше, чем создал. А вот счётчик блокировок может перейти границу, если клиент вызовет IClassFactory.LockServer(False) больше раз, чем это нужно. Вообще говоря, реализация ICalssFactory.LockServer должна учитывать это и не давать возможности счётчику блокировок становиться отрицательным.**

Процесс выгрузки неиспользуемых библиотек не автоматизирован полностью. Чтобы система выгрузила библиотеки, в которых клиент перестал нуждаться, последний должен вызвать функцию **CoFreeUnusedLibraries**. Те **COM-объекты**, которые используются в рамках одной нити, выгружаются немедленно. Для многонитевых серверов

DllCanUnloadNow вызывается дважды: сначала создаётся список всех библиотек, которые считают себя готовыми к выгрузке, затем через 10 минут для библиотек из этого списка снова вызывается **DllCanUnloadNow**. Те библиотеки, функции которых снова вернут **S_OK**, выгружаются из памяти. Такая задержка нужна для того, чтобы библиотека могла корректно завершить созданные ею нити. Если вас не устраивает десятиминутный интервал, можно использовать функцию **CoFreeUnusedLibrariesEx** — с её помощью этот интервал можно устанавливать явно.

Для корректной работы сервер должен быть зарегистрирован в системном реестре. В общем случае в реестр заносится довольно много записей, которые зависят от различных особенностей реализации сервера.

В нашем случае функция **DllRegisterServer** производит нужные записи в системный реестр.

Функция **DllUnregisterServer** должна удалить из реестра всё, относящееся к данному COM-серверу. В идеале эта функция должна учитывать то, что могут существовать и другие версии этого сервера, которые будут использовать часть "наших" ключей, проверять это и удалять только то, что действительно не нужно больше никому.

Для регистрации внутреннего COM-сервера и её отмены существует специальная системная утилита **RegSvr32.exe**. Чтобы зарегистрировать сервер, реализованный в виде **DLL**, нужно запустить эту утилиту, передав ей в качестве параметра командной строки имя библиотеки. Утилита найдёт в этой библиотеке функцию **DllRegisterServer** и запустит её, в результате чего сервер будет зарегистрирован. Чтобы отменить регистрацию, нужно к командной строке **RegSvr32.exe** добавить ещё один ключ — "/u". В этом случае утилита будет искать и запускать функцию **DllUnregisterServer**.

На этом создание нашего простейшего COM-сервера завершено. Можно его откомпилировать и зарегистрировать получившуюся библиотеку. После этого можно писать для неё клиент.

Для обращения к серверу создадим клиент, состоящий из двух модулей — **Test_Main.dpr** и **ClientTest_Client.pas**. Кроме того, он использует файл **Test_Interface.pas** и модуль **ComObj**. Главное окно клиента представлено на рисунке 4.1.

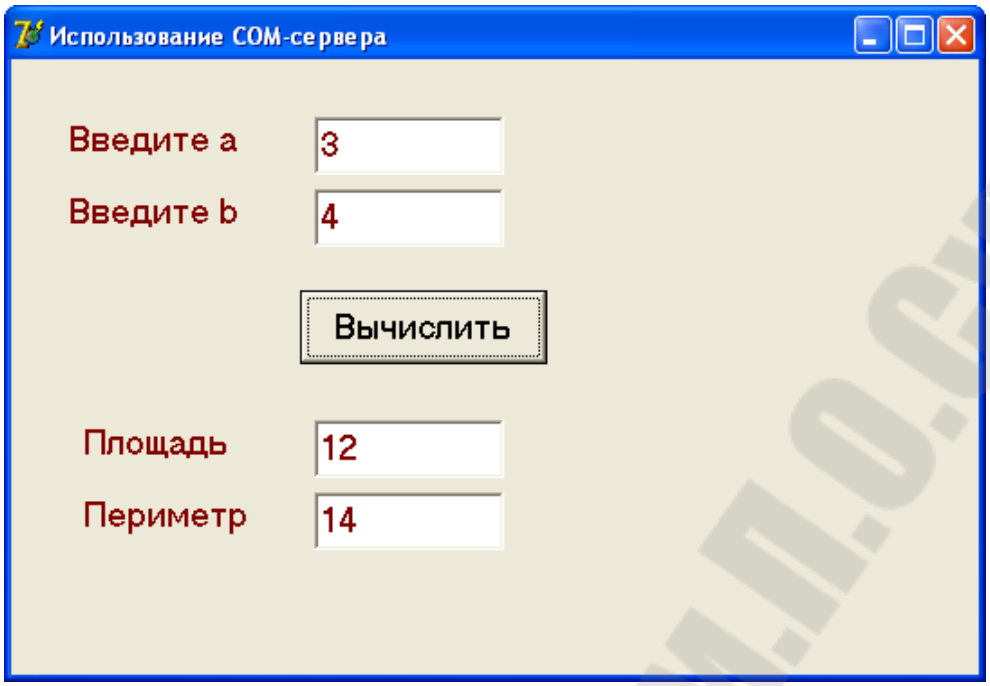

Рисунок 4.1

Клиент при запуске создаёт экземпляр сервера, который мы только что написали. Элементы управления позволяют задавать аргументы для методов сервера и вызывать их.

Код модуля клиента представлен ниже.

```
unit Test_Client;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls, ComObj, Test_Interface;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    Label3: TLabel;
    Label4: TLabel;
```

```
    Edit3: TEdit;
    Edit4: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Itest:ItestServer;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
Var Res: HResult;
Pl,Per:real;
begin
   ITest := CreateComObject(CLSID_TestServer) as ITestServer;
   if ITest = nil then
  begin
    ShowMessage('Сервер не подключен');
    Exit;
  end;
  Res := ITest.setarg(StrToInt(Edit1.Text),StrToInt(Edit2.Text));
  Res:=Itest.Plosh(Pl);
  Res:=Itest.Per(Per);
  edit3.Text:=FloatToStr(Pl);
  edit4.Text:=FloatToStr(Per);
end;

end.
```

Как видите, нам удалось создать внутренний COM-сервер, с которым клиент может работать стандартным способом (с учётом оговоренных в самом начале ограничений).

4.2 Создание COM-сервера средствами Delphi

Рассмотрим создание простейшего COM-сервера в среде Delphi на примере перевода целого десятичного числа в двоичный, восьмеричный и шестнадцатеричный вид.

Для создания встроенного сервера (реализуется в виде DLL) выбираем в меню **File/New**. В появившемся диалоговом окне **New Item** открываем закладку **ActiveX** и выбрать элемент **ActiveX Library**.

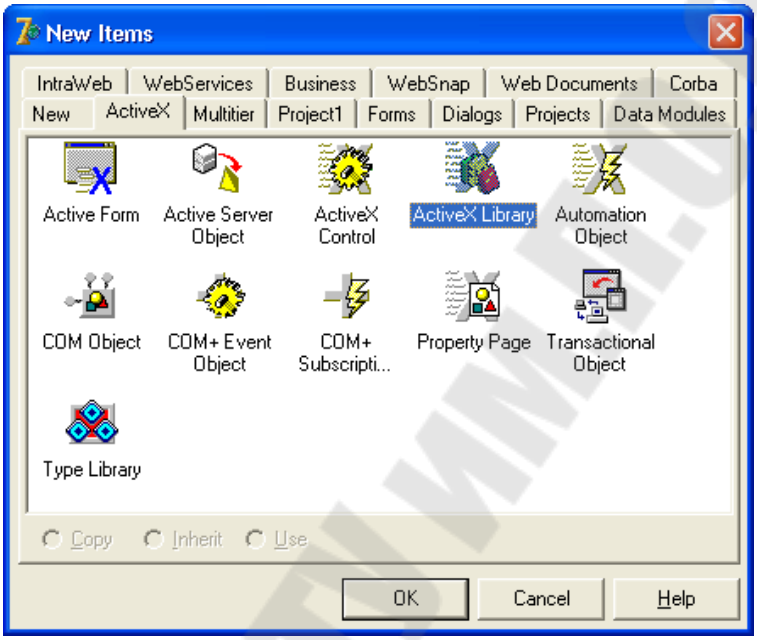

Рисунок 4.2

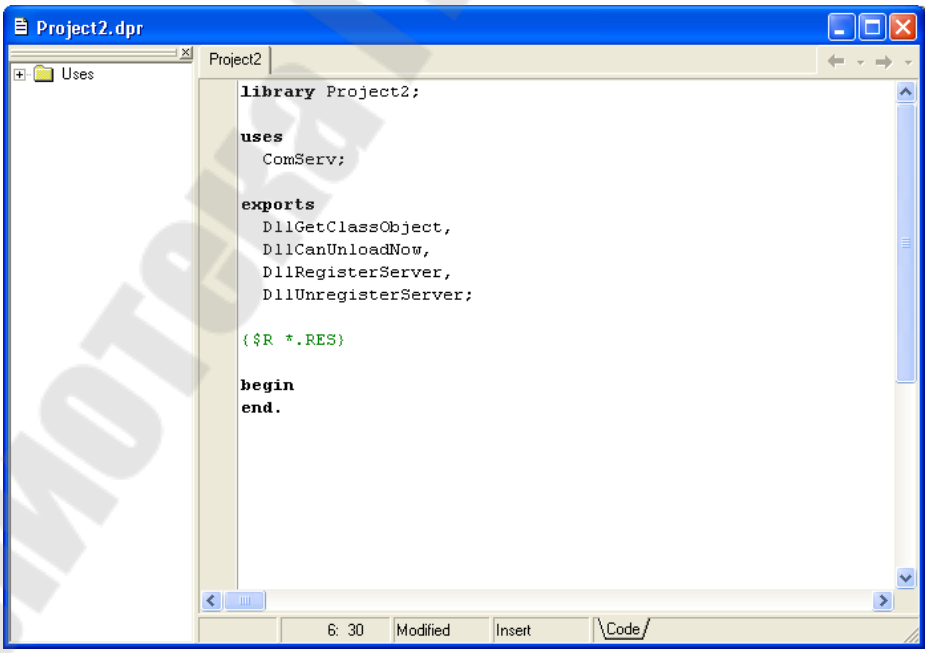

Рисунок 4.3

В библиотеку автоматически прописываются четыре экспортируемые функции, которые реализованы в модуле **ComServ**:

DllRegisterServer – используется для регистрации Com-сервера в системном реестре. Если регистрация прошла успешно, функция возвращает значение S_OK, иначе S_FAIL.

DllUnregisterServer – используется для удаления из системного реестра всех компонентов, относящихся к данному Com-серверу. Если функция отработала успешно, то возвращается значение S_OK, иначе S_FAIL.

DllGetClassObject – создает новый компонент с номером CLSID и выдает ссылку на интерфейс с номером IID через параметр Obj. В случае успешноговыполнения функция возвращает значение S_OK, иначе E_NOINTERFACE (интерфейс не найден) или CLASS_E_CLASSNOTAVAILABLE (компонент с таким номером не зарегистрирован в системе).

DllCanUnloadNow – эта функция вызывается механизмом реализации Com для проверки , выполняются ли удаления Com-сервера из памяти. Если какое-то приложение имеет ссылку на любой компонент сервера, то функция вернет S_FALSE, в противном случае S_TRUE, при этом Com-сервер будет удален из паямти.

После создания Com-сервера необходимо разработать интерфейс, который является связующим звеном между клиентом и сервером.

Для создания интерфейса добавим в проект новый модуль и сохраним под именем **server_interface.pas**. В тексте модуля описываем интерфейс

```
IConvert=Interface
  ['{738810F5-FE8D-49C0-B44B-4EA81B5AC167}']
  function Bin(const N:Word; var S:WideString):Boolean; stdcall;
  function Oct(const N:Word; var S:WideString):Boolean; stdcall;
  function Hex(const N:Word; var S:WideString):Boolean; stdcall;
  end;
```

Для генерации **GUID** используем сочетание клавиш Ctrl+Shift+G.

Теперь необходимо добавить компоненты (хотя бы один). Для этого снова открываем окно, показанное на рисунке 4.2, и выбираем COM Object. После этого запускается эксперт, который отображает окно, показанное на рисунке 4.3.

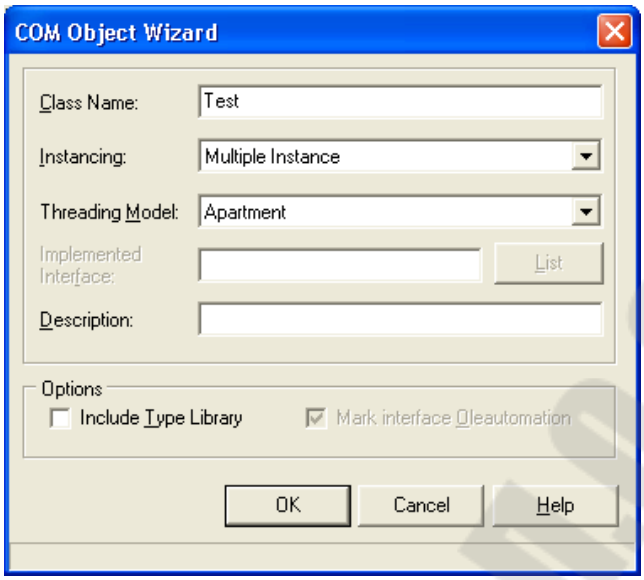

Рисунок 4.3

На рисунке 4.3 поля заполнены именно так, как необходимо для нашего простейшего COM-сервера.

Поле ClassName — это имя кокласса. Имя реализующего его класса Delphi образуется из этого имени добавлением префикса "T".

Поле Instancing управляет тем, как будет создаваться COM-объект. Имеются три варианта: **Internal**, **Single Instance** и **Multiple Instance**. COM-объект, создающийся как **Internal** — это тот самый несоздаваемый объект, про который мы уже говорили. Он не регистрируется в реестре и не может быть создан непосредственно клиентом. Если объект создаётся как **Single Instance**, каждый клиент будет работать со своей копией сервера. Это обеспечивается флагом **RegCls_SingleUse** в параметрах функции **CoRegisterClassObject**: если этот флаг установлен, то сразу после того как один из клиентов получит указатель на зарегистрированную таким образом фабрику класса, только этот клиент и будет её видеть. Для всех остальных приложений фабрика класса перестанет быть зарегистрированной, поэтому попытка другого клиента использовать такой же COM-объект приведёт к запуску новой копии сервера. И, наконец, выбранный по умолчанию вариант **Multiple Instance** означает, что фабрика класса будет доступна всем клиентам независимо от того, кто из них первый её использовал. Что касается внутреннего **COM-сервера**, то он, во-первых, не использует **CoRegisterClassObject**, а во-вторых, такой сервер в силу внутренней природы **DLL** не может работать одновременно с несколькими клиентами, поэтому для него разницы между **Single Instance** и **Multiple Instance** нет.

Следующее поле — **Threading Model** (нитевая модель). Это довольно сложное понятие. Отметим, что нитевая модель — это своего

рода контракт, заключаемый между **COM-объектом** и операционной системой по синхронизации в случае распараллеливания работы на несколько нитей. Этот контракт определяет разграничение ответственности — в каких случаях сервер самостоятельно реализует синхронизацию доступа к данным из разных нитей, в каких это за него делает система. Во многих частных случаях сервер может синхронизировать доступ к данным гораздо эффективнее, чем это делает система, использующая универсальные методы, но такая эффективность требуется не всегда, что позволяет упрощать сервер, перекладывая задачу синхронизации на систему. В нашем случае мы пока не предоставляем системе средства, с помощью которых она могла бы выполнить свою часть контракта (именно поэтому клиент должен работать с сервером только из главной нити), так что нам подходит любой из предложенных для этого поля вариантов, кроме **Free**.

Поле **Description** содержит простое описание кокласса, которое используется при регистрации в реестре. И, наконец, очень важный момент — галочка **Include Type Library** должна быть снята. **Библиотека типов** — это очень мощный инструмент, и мы рассмотрим её позже, но пока наш сервер обойдётся без неё.

После заполнения всех полей эксперт создаёт новый модуль вот с таким кодом:

```
unit Unit1;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj;

type
  TTest= class(TComObject, IConvert)
  protected
    {Declare IConvert methods here}
  end;

const
  Class_Test:      TGUID      =      '{3A064EE6-B0E4-4FBF-9BD3-
C779BA091D32}';
```

```
implementation

uses ComServ;

initialization
  TComObjectFactory.Create(ComServer, TTest, Class_Test, 'Test', '',
ciMultiInstance, tmApartment);
end.
```

Эксперт автоматически создал заготовку для нашего кокласса, константу, содержащую его **CLSID** (которая начинается с префикса Class вместо традиционного **CLSID**), а в раздел инициализации вставил создание фабрики класса (причём уже отсюда видно, что фабрика класса создаётся только один раз при запуске сервера, а не по запросам клиента).

Поддержку интерфейса **IConvert** придётся добавлять вручную. Для этого, во-первых, добавим модуль **server_interface** в раздел **uses**. Затем переделаем объявление класса **TTest**, дописываем реализацию функций интерфейса и сохраняем под именем **test_object.pas**

```
unit test_object;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj,server_interface;

type
  TTest = class(TComObject,IConvert)
  protected
  function Bin(const N:Word; var S:WideString):Boolean; stdcall;
  function Oct(const N:Word; var S:WideString):Boolean; stdcall;
  function Hex(const N:Word; var S:WideString):Boolean; stdcall;
  end;

const
  Class_Test: TGUID = '{37D69CC7-7971-4FD8-8FAC-
96E34C5E4A63}';

implementation
```

```
uses ComServ;


function TTest.Bin(const N:Word; var S:WideString):Boolean;
var S1:string;
   m:Word;
begin
result:=true;
try
S1:='';
m:=n;
repeat
S1:=chr((m mod 2)+48)+S1;
m:=m div 2;
until m=0;
S:=WideString(S1);
except
result:=false;
end;
end;

function TTest.Oct(const N:Word; out S:WideString):Boolean;
var S1:string;
   m:Word;
begin
result:=true;
try
S1:='';
m:=n;
repeat
S1:=chr((m mod 8)+48)+S1;
m:=m div 8;
until m=0;
S:=WideString(S1);
except
result:=false;
end;
end;

function TTest.Hex(const N:Word; out S:WideString):Boolean;
var S1:string;
   m,k:Word;
```

```
begin
result:=true;
try
S1:='';
m:=n;
repeat
S1:=chr((m mod 16)+48)+S1;
k:=m div 16;
if k<10 then S1:=chr(k+48)+S1
      else S1:=chr(k+57)+S1;
m:=m div 16;
until m=0;
S:=WideString(S1);
except
result:=false;
end;
end;


initialization
  TComObjectFactory.Create(ComServer, TTest, Class_Test,
    'Test', '', ciSingleInstance, tmApartment);
end.
```

На этом создание простого сервера средствами Delphi завершено (выбираем команду **Project – Build Project1**), осталось только зарегистрировать его в системе (с помощью утилиты **RegSvr32.exe**).

Теперь создадим клиента, который будет подключаться к серверу. Для этого создаем новое приложение, подключаем дополнительно модули **ComObj, server_interface**

```
unit client_for_server;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  StdCtrls, ComObj, server_interface,dialogs;

const
```

```
    Class_Test: TGUID = '{37D69CC7-7971-4FD8-8FAC-
96E34C5E4A63}';
  type
   TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    procedure Button4Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
   private
    { Private declarations }

    test1:IConvert;
   public
    { Public declarations }
   end;

  var
   Form1: TForm1;

  implementation

  {$R *.dfm}

  procedure TForm1.Button4Click(Sender: TObject);
  Var t1:TGUID;
  begin
  t1:= Class_Test;
  showmessage(GUIDToString(t1));

  test1 := CreateComObject(t1) as Iconvert;
   if test1= nil then
     ShowMessage('Сервер не подключен')
   else ShowMessage('Сервер подключен');
  end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
Var N:word;
Str:WideString;
begin
N:=StrToInt(Edit1.Text);
if test1.Bin(N,Str) then ShowMessage(Str);

end;

procedure TForm1.Button2Click(Sender: TObject);
Var N:word;
Str:WideString;
begin
N:=StrToInt(Edit1.Text);
if test1.Oct(N,Str) then ShowMessage(Str);
end;

procedure TForm1.Button3Click(Sender: TObject);
Var N:word;
Str:WideString;
begin
N:=StrToInt(Edit1.Text);
if test1.Hex(N,Str) then ShowMessage(Str);
end;

end.
```

Пример подключения COM-клиента приведен на рисунке 4.4

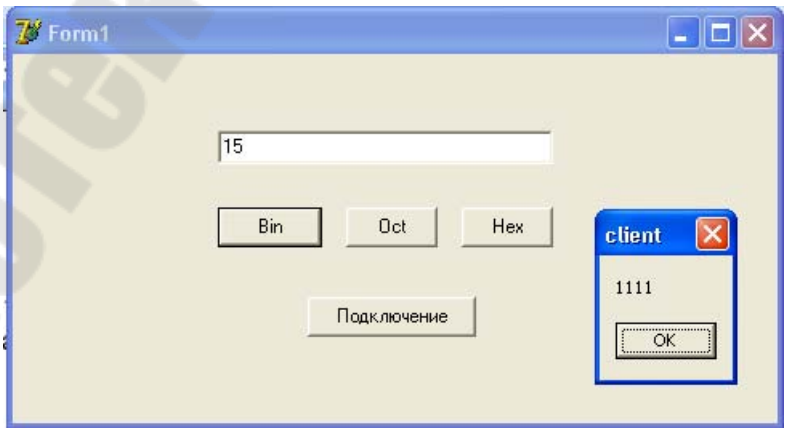

Рисунок 4.4

На этом простом примере хорошо видно, от скольких проблем избавляют разработчика средства Delphi: фактически, единственное, что нам здесь пришлось программировать — это поддержку специфичного для нашего сервера интерфейса **IConvert**, а всё то, что составляет основу любого COM-сервера, уже реализовано в модулях **ComObj** и **ComServ.**

4.4 Создание COM-сервера с использованием библиотеки типов

4.4.1 Библиотека типов

Библиотека типов – это специальный двоичный ресурс, описывающий интерфейсы и методы, реализуемые COM-сервером.

Библиотека типов представляет собой список фабрик классов, поддерживаемых данным COM-сервером. Для каждой фабрики классов можно получить список интерфейсов. Для каждого интерфейса приводится список поддерживаемых методов, для каждого из которых можно затребовать список формальных параметров. Порядок следования методов в списке соответствует их порядку следования в виртуальной таблице методов — поэтому на этапе компиляции клиента можно связать вызов метода сервера с соответствующим столбцом виртуальной таблицы.

Все эти списки сохраняются либо непосредственно в **COM-сервере** (в файлах с расширениями *.exe, *.dll, *.ocx), либо отдельно (в файлах с расширениями *.tlb, *.olb). Для их хранения предусмотрены специальные двоичные форматы. Соответственно, инструменты программирования, поддерживающие импорт библиотек типов, должны знать форматы этих файлов, уметь читать данные из них и представлять эти данные разработчику в естественном, специфическом для данного языка программирования виде. Например, если библиотека типов импортирована в средство разработки, использующее компилятор C++, то ее реализация должна быть представлена с помощью синтаксиса C++; если импортирована в Delphi — с помощью синтаксиса Delphi. При этом разработчик должен отдавать себе отчет, что реально библиотека типов хранится совсем в других форматах. Если же описание библиотеки типов хранится непосредственно в **COM-сервере**, то на сервере можно реализовать интерфейс **ITypelnfo**. Пользуясь методами этого интерфейса, среда разработки может опросить сервер о тех же самых параметрах, что считываются из вышеперечисленных файлов. Пользоваться интерфейсом ITypelnfo гораздо проще, чем писать конверторы форматов, и опытный разработчик может легко узнать о содержимом библиотеки типов сервера в клиентском приложении. Соответственно, сама библиотека типов в этом случае может храниться в сервере в произвольном формате.

Для создания и редактирования библиотек типов в Delphi имеется редактор, совмещенный с редактором интерфейсов (определение фабрик классов, интерфейсов и их методов). Кроме наличия библиотеки типов сервер должен поддерживать интерфейс **IProvideClassInfo**. В Delphi такой сервер реализуется путем наследования его от **TTypedComObject**. Для этого необходимо оставить включенным флаг **Include Type Library** в мастере создания **COM-объекта**.

4.4.2 Язык IDL

Многие средства разработки, используемые для создания **COM-серверов**, содержат в комплекте поставки утилиты для автоматической генерации библиотек типов или клиентского и серверного кода на основании описаний интерфейсов сервера.

Во многих случаях эти описания создаются на языке **IDL** (**Interface Definition Language** — язык определения интерфейсов).

Язык **IDL** предназанчен для спецификации интерфейсов. Фактически это стандарт, позволяющий описывать вызываемые методы сервера и их параметры, не вдаваясь в детали и правила реализации серверов и клиентов на том или ином языке программирования. Используя **IDL**, можно описать интерфейсы сервера, а затем создать его реализацию, равно как и реализацию клиента, на любом языке программирования с помощью широкого спектра средств разработки. В определенном смысле **IDL** — это стандарт для описания взаимодействия между компонентами распределенной системы, не зависящий от деталей реализации и языков программирования (а в общем случае не зависящий также от платформ, поскольку IDL используется не только в COM).

Язык **IDL** в технологии **COM** является преемником этого языка в технологии **DCE** (**Distributed Computing Environment** — среда распределенных вычислений) — спецификации межплатформенного взаимодействия служб, разработанной консорциумом **Open Systems Foundation**. Отметим, что в настоящее время существует несколько диалектов **IDL (для COM, для CORBA, для DCE и др.).** Тем не менее различия между ними невелики.

Язык **IDL** немного похож па язык **C++** в той его части, которая относится к описанию классов (то, что обычно помещается в h-файлы). В качестве примера приведем описание на **IDL** интерфейсов гипотетического **COM-сервера**, содержащего объект **MyComObj**, интерфейс которого **IMyComObj** (наследник **IUnknown**) экспонирует два метода: **метод MyMethodl**, получающий в качестве входных параметров два целых числа и возвращающий действительное число, и **метод**

MyMethod2, не возвращающий данных и получающий в качестве входного параметра переменную типа **Variant**:

```
[
uuid(845256D0-8E96-llD2-B126-000000000000).
version(l.O),
helpstring("Projectl Library")
]
library Projectl
{
importlib("STD0LE2.TLB"):
importlib("STDVCL40.DLL"):
[
uuid(845256Dl-8E96-llD2-B126-000000000000).
version(l.O).
helpstring("Interface for myComObj Object")
]
interface ImyComObj: IUnknown
{
[id(OxOOOOOOOl)]
double _stdcall MyMethodl([in] long Paraml,
[in] long Param2 ):
[id@x00000002)]
void _stdcall MyMehod2([in] VARIANT Рагат3 ):
}:
[
uuid(845256D3-8E96-llD2-B126-000000000000).
version(l.O),
helpstring("myComObj Object")
]
coclass myComObj
{
[default] interface ImyComObj;
};
}:
```

Многие средства разработки, поддерживающие создание **COM-серверов**, имеют в своем составе утилиты для генерации серверного и клиентского кода на основании описаний на **IDL**. В частности, в состав **Microsoft Visual C++** включен компилятор **MIDL**, генерирующий код для клиентских и серверных библиотек, ответственных за взаимодействие

клиента и сервера, па основании созданных разработчиками описаний на **IDL**.

Отметим, однако, что при создании COM-серверов **с помощью мастеров Delphi библиотеки типов и соответствующий код (или его «заготовки») генерируются автоматически, и нет необходимости вручную создавать описания на IDL**. При этом всегда можно сгенерировать описание на **IDL** на основании библиотеки типов сервера, созданной с помощью соответствующего редактора, представленного на рисунке 4.5.

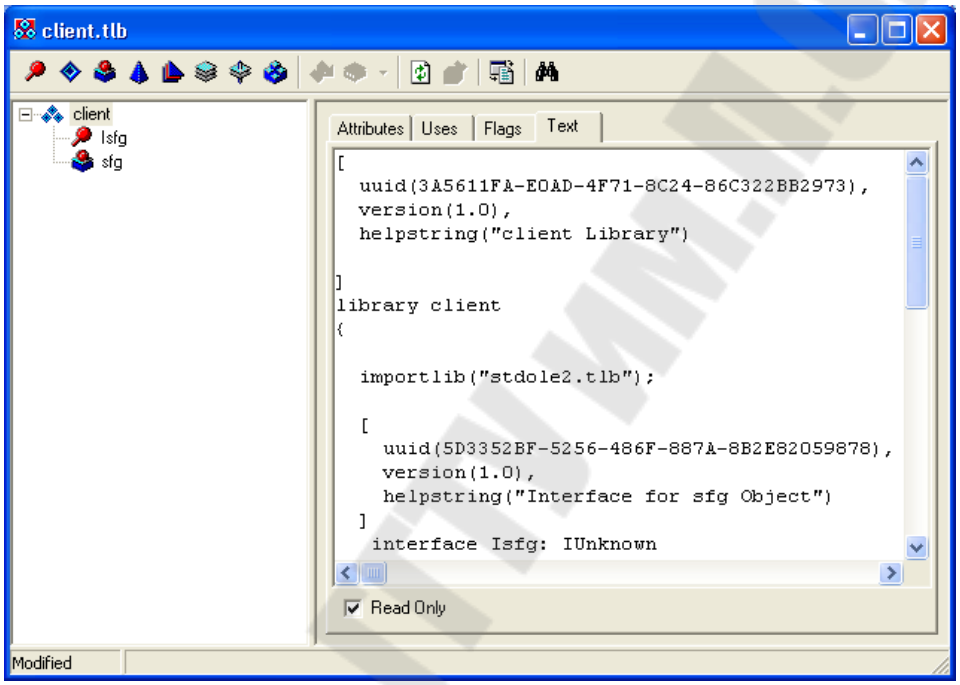

Рисунок 4.5

4.4.3 Создание COM-сервера

Для создания встроенного сервера (реализуется в виде DLL) выбираем в меню **File/New**. В появившемся диалоговом окне **New Item** открываем закладку **ActiveX** и выбираем элемент **ActiveX Library** (рисунок 4.2).

Для создания COM-объекта выбираем в меню **File/New**. В появившемся диалоговом окне **New Item** открываем закладку **ActiveX** и выбираем элемент **COM Object**. В появившемся диалоговом окне вводим название **КоКласса**, в опциях устанавливаем флаг **Include Type Library**, флаг **Mark interface Oleautomation** снимаем (рисунок 4.3).

После нажатия ОК открывается редактор библиотеки типов (рисунок 4.7)

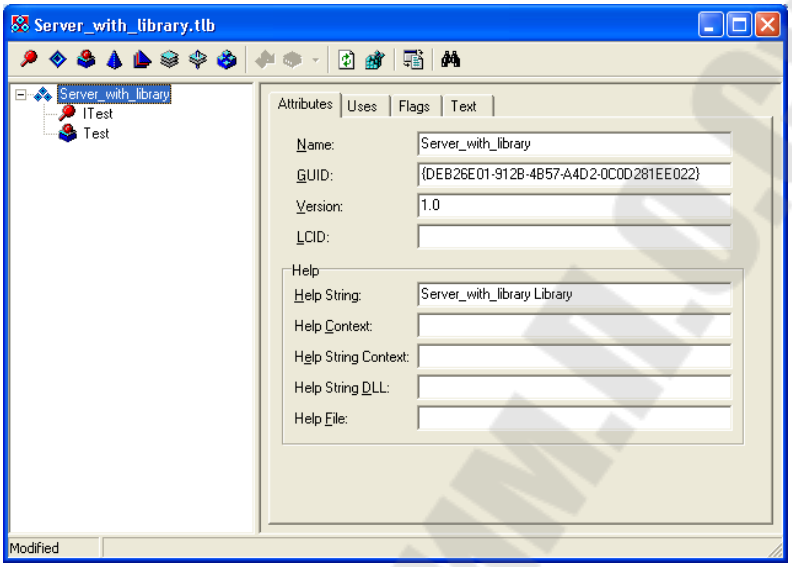

Рисунок 4.7

Выбираем интерфейс **ITest** и с помощью контекстного меню (или кнопки на панели инструментов) выбираем команду **New Method** (рисунок 4.8).

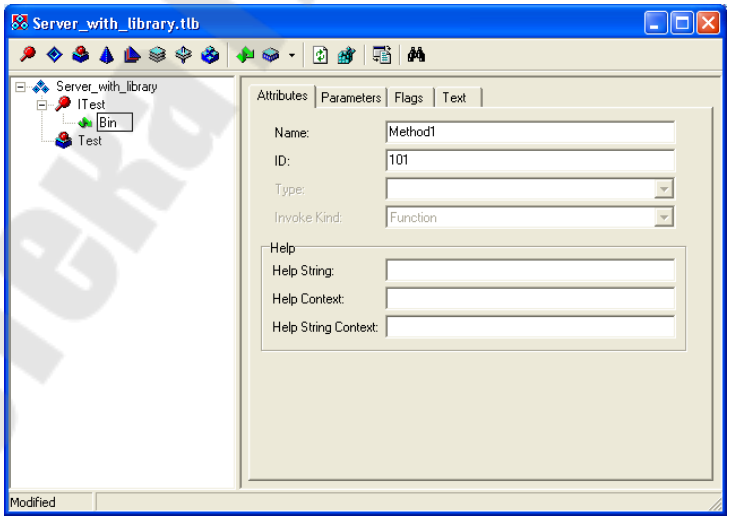

Рисунок 4.8

Параметры метода находятся на вкладке **Parameters**. Каждый параметр имеет имя, тип и набор модификаторов — флагов и атрибутов параметра. Параметры могут иметь следующие модификаторы:

Таблица 2

Модификатор	Описание
In	Параметр является входным, т.е. через него вызывающая сторона передаёт данные вызываемой.
Out	Параметр является выходным, т.е. через него вызываемая сторона возвращает данные вызывающей. Параметр должен быть указателем (т.е. значение передаётся по ссылке). Допускается указание для одного параметра модификаторов In и Out одновременно — в этом случае параметр является и входным, и выходным. Модификаторы In и Out очень важны для правильного маршалинга: без них библиотека oleaut32.dll не сможет правильно передавать параметры через границы процесса, так что забывать об этих модификаторах нельзя.
RetVal	Через данный параметр возвращается значение функции (этот модификатор допустим только для Out-параметров). Как мы уже говорили, для полной совместимости с COM/DCOM методы интерфейса должны возвращать значение типа HRESULT. Но некоторые языки программирования (например, Visual Basic) позволяют вызывать методы, имеющие параметр с модификатором RetVal так, как будто это функции, возвращающие RetVal-значение. RetVal-параметр должен быть последним в списке параметров.
LCID	Параметр является идентификатором языка. Если метод поддерживает несколько языков (имеются ввиду естественные языки, а не языки программирования), через этот параметр передаётся идентификатор языка. Параметр с данным модификатором должен иметь тип long и стоять в списке параметров либо последним, либо предпоследним непосредственно перед RetVal-параметром. Смысл данного модификатора в том, что некоторые среды программирования позволяют не

	указывать данный параметр явно, а автоматически при вызове подставляют идентификатор текущего языка.
Optio□al	Параметр является необязательным. Такой параметр должен иметь тип VARIANT или VARIANT*. Некоторые среды разработки позволяют при вызове метода не указывать необязательные параметры, передавая вместо них пустое (т.е. VT_EMPTY) значение.
Has Default Val□e	Параметр имеет значение по умолчанию (при выборе этого модификатора становится доступно поле ввода значения по умолчанию). По сути, то же самое, что и Optional-параметр, только вместо пустого значения передаётся заранее заданное. Параметры со значением п□ умолчанию могут иметь не только тип VARIANT, но и любой VARIANT-совместимый тип.

На вкладке Parameters (рисунок 4.9) определяем входные параметры соответствующего метода.

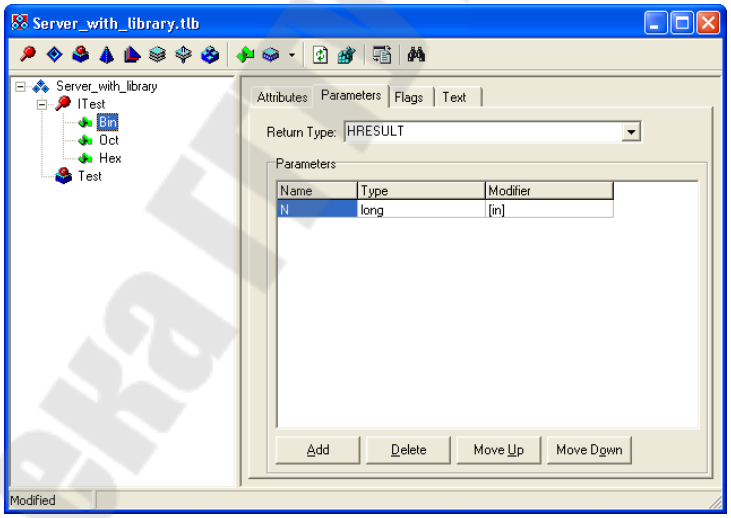

Рисунок 4.9

Создаем методы **Oct** и **Hex** аналогичным образом. После этого нажимаем кнопку **Refresh (Обновить)** для записи методов в код модуля. Обратите внимание, что к модулю **Com-объекта** автоматически добавляются модули **ComObj, Server_with_library_TLB**. Последний является описанием библиотеки типов.

В разделе **implementation** модуля **Com-объекта** необходимо описать реализацию методов

```
function TTest.Bin(N: Integer): HResult;
var S:String;
   m:Word;
begin
S:='';
m:=N;
repeat
S:=chr((m mod 2)+48)+S;
m:=m div 2;
until m=0;
showmessage(widestring(s));
Result := S_OK;
end;

function TTest.Hex(N: Integer): HResult;
var S:string;
   m,k:Word;
begin
S:='';
m:=N;
repeat
S:=chr((m mod 16)+48)+S;
k:=m div 16;
if k<10 then S:=chr(k+48)+S
       else S:=chr(k+57)+S;
m:=m div 16;
until m=0;
showmessage(widestring(s));
Result := S_OK;
end;

function TTest.Oct(N: Integer): HResult;
var S:string;
   m:Word;
begin
S:='';
m:=N;
repeat
S:=chr((m mod 8)+48)+S;
```

```
m:=m div 8;
until m=0;
showmessage(widestring(s));
Result := S_OK;
end;
```

Для создания COM-сервера запускаем процесс компиляции или выбираем команду **Project – Build server_whith_library**. **Для регистрации сервера в редакторе библиотеки типов выбираем команду Register Type Library.**

Рассмотрим последовательность действий для создания COM-клиента для созданного сервера.

1. Создаем новое приложение (рисунок 4.10)

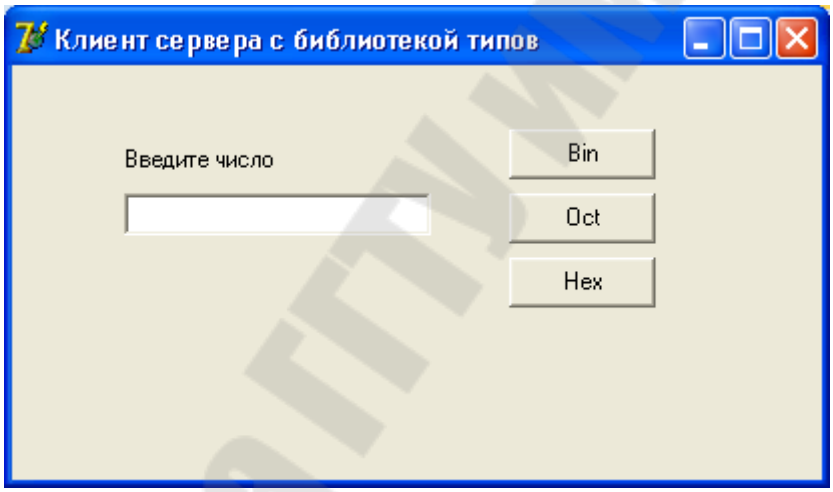

Рисунок 4.10

2. Выбираем команду **Project – Import Type Library** (рисунок 4.11)
.

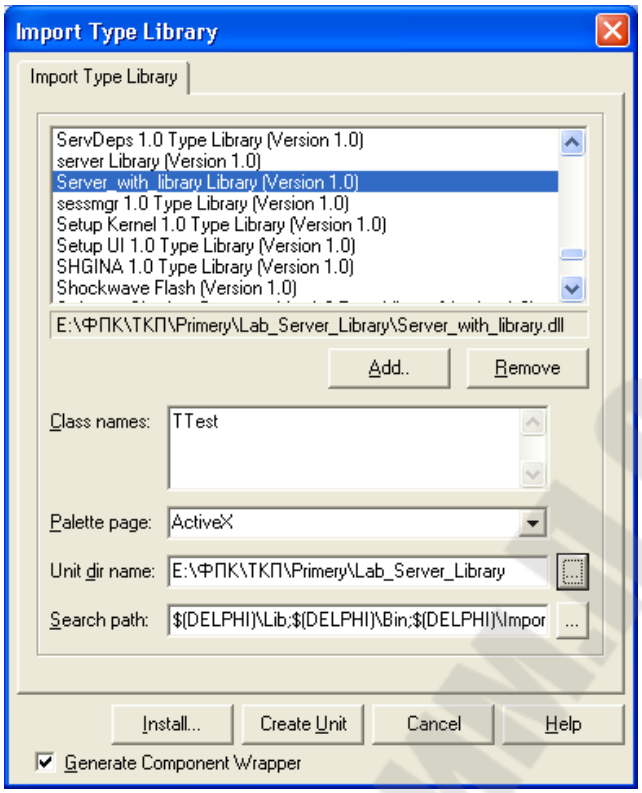

Рисунок 4.11

В диалоговом окне выбираем соответствующую библиотеку. Прописываем путь для записи файла библиотеки типов с модулем **Com-клиента**. Снимаем флаг с **Generate Component Wrapper**. Нажимаем кнопку **Create Unit**. В окне **Com-клиента** подключаем библиотеку типов и описываем последовательность действий клиента.

```
procedure TForm1.Button1Click(Sender: TObject);
Var t1:TGUID;
N:integer;
begin
t1:= Class_Test;
test2 := CreateComObject(t1) as ITest;
  if test2= nil then
    ShowMessage('сервер не подключен')
 else ShowMessage('сервер подключен');
N:=strtoInt(Edit1.Text);
test2.Bin(N);

end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
Var t1:TGUID;
N:integer;
begin
t1:= Class_Test;
test2 := CreateComObject(t1) as ITest;
  if test2= nil then
    ShowMessage('сервер не подключен')
 else ShowMessage('сервер подключен');
N:=strtoInt(Edit1.Text);
test2.Oct(N);

end;

procedure TForm1.Button3Click(Sender: TObject);
Var t1:TGUID;
N:integer;
begin
t1:= Class_Test;
test2 := CreateComObject(t1) as ITest;
  if test2= nil then
    ShowMessage('сервер не подключен')
 else ShowMessage('сервер не подключен");
N:=strtoInt(Edit1.Text);
test2.Hex(N);
end;
```

Таким образом, для распространения и использования сервера не нужно ничего, кроме его исполнимого модуля. Но самое главное даже не это. Гораздо более важно, что Вы можете импортировать и использовать в своей программе любой из имеющихся на компьютере COM-серверов. Естественно, что при передаче своей программы клиенту Вы должны установить на его компьютере соответствующий COM-сервер.

5 Маршалинг

Мы уже говорили, что местонахождение сервера прозрачно для клиента, но не касались вопроса о том, как эта прозрачность обеспечивается. Ведь, по сути дела, клиент вызывает методы, которые могут располагаться в адресном пространстве другого процесса или даже на удалённом компьютере. На самом деле методы, естественно, вызываются в адресном пространстве сервера. На клиентской стороне собирается вся информация, необходимая для вызова метода, затем эти данные передаются серверу, расшифровываются, вызывается нужный метод с нужными параметрами, а результат передаётся клиенту в обратном порядке. Таким образом, задача вызова метода через границы процесса разбивается на три части:

1. Сбор на стороне клиента всех необходимых для вызова данных и сериализация их в единый поток. Этот этап называется **маршалингом (marshaling)**.
2. Передача сформированного потока через границы процесса или по сети.
3. Извлечение на стороне сервера данных из потока и формирование на их основе структур, которые затем будут переданы методу в качестве параметров. Этот этап называется **обратным маршалингом (unmarshaling**).

Нередко под словом "маршалинг" объединяют все три этапа процесса, хотя это не совсем правильно.

Детальное знакомство с этапами вызова мы начнём со второго из них. Это единственный этап, на котором не учитывается специфика вызываемого метода. Все действия системы на данном этапе сводятся к решению простой транспортной задачи: как передать поток заданной длины из адресного пространства клиента в адресное пространство сервера. Эта задача решается специальными системными объектами, которые называются **объектами канала**. Объект канала на стороне клиента получает поток данных и передаёт его объекту канала на стороне сервера. Передача осуществляется с помощью протокола **ORPC (Object Remote Procedure Call — вызов удалённых процедур объекта).**

Разработчики **COM/DCOM** многое позаимствовали из стандарта **DCE RPC (Distributed Computing Environment Remote Procedure Call)**. Этот стандарт, разработанный **OSF** (Open Software Foundation), предназначен для вызова процедур, реализованных на удалённом компьютере. На нём основана такая широко известная модель как **CORBA**. При вызове методов удалённого объекта объекты канала используют **RPC** в чистом виде — как уже было сказано, специфика методов на данном

этапе значения не имеет, поэтому **RPC** одинаково хорошо подходит для удалённого вызова как обычных процедур, так и методов **COM-объектов**. При вызове методов локального сервера используется разработанный Microsoft специально для **COM/DCOM** протокол **LRPC (Lightweight RPC — облегчённый RPC)**, который близок к **RPC**, но оптимизирован для передачи потока через общую область памяти, а не через сеть. Протоколы **RPC** и **LRPC** вместе называются **ORPC**. Чаще всего **RPC** для передачи данных использует протокол **TCP**, хотя системные настройки позволяют использовать и другие протоколы. В любом случае ни клиент, ни сервер об этом не заботятся — система использует требуемый транспортный протокол самостоятельно.

Теперь вернёмся к первому и третьему этапам — **прямому и обратному маршалингу.** Этим занимаются два специальных объекта — **заместитель (proxy) и заглушка (stub)**. Заместитель загружается в адресное пространство клиента. Упрощённо его можно рассматривать как специальный COM-объект, реализующий те же интерфейсы, что и тот объект, который клиент использует. Фактически, клиент работает с интерфейсами заместителя, а не самого объекта. Когда клиент думает, что он вызывает метод COM-объекта, на самом деле он вызывает соответствующий метод заместителя. Заместитель, получив вызов, формирует поток, который следует отослать серверу, и передаёт его своему объекту канала. До этого момента данные передаются в пределах одного адресного пространства. Далее клиентский объект канала, используя ORPC, передаёт поток серверному объекту, а тот — заглушке (и серверный объект, и заглушка находятся в адресном пространстве сервера). Заглушка извлекает данные из потока, восстанавливая их структуру, и вызывает соответствующий метод сервера. После завершения этого метода результаты его работы передаются клиенту по этой же цепочке, но в обратном порядке. **Таким образом, для входных параметров метода прямой маршалинг выполняется заместителем, а обратный — заглушкой, а для выходных параметров — наоборот.**

Использование заглушки и заместителя приводит к тому, что клиент всегда вызывает методы заместителя в своём адресном пространстве, а сервер получает вызовы от заглушки также в своём адресном пространстве. Процесс передачи данных из одного адресного пространства в другое невидим ни для сервера, ни для клиента.

Упаковка данных в поток и их последующее извлечение — не такая простая задача, как это может показаться на первый взгляд. Основная проблема заключается в передаче параметров. С параметрами простых типов (Integer, Real, Boolean и т.п.) всё просто — они в нужном порядке заносятся в поток и так же из него извлекаются. А вот с указателями так просто не получится: бессмысленно передавать в поток само значение

указателя — в другом адресном пространстве оно не будет иметь никакого смысла. Нужно вставить в поток ту область памяти, на которую ссылается указатель. Для этого надо знать размер этой области. Кроме того, указатель может ссылаться на структуру, которая, в свою очередь, тоже может содержать указатели, которые также нужно передавать не по значению, а копировать в поток нужную область памяти. Некоторые специфические типы (как, например, BSTR) могут хранить данные и в области, находящейся по отрицательному смещению от указателя. Чтобы корректно упаковать все данные в поток, а затем извлечь их и восстановить их структуру, и заглушка, и заместитель должны обладать исчерпывающей информацией о всех типах данных, используемых параметрами. Кроме того, они должны знать, какие параметры являются входными и передаются от клиента серверу, а какие — выходными (от сервера клиенту). Очевидно, что построение универсальных заместителя и заглушки, в равной мере пригодных для всех интерфейсов, невозможно.

В COM/DCOM существует три вида маршалинга, различающиеся тем, кто несёт ответственность за упаковку и распаковку данных:

1. **Стандартный маршалинг.** В этом случае заместитель и заглушка создаются специальной динамически компонуемой библиотекой, которая так и называется — **proxy/stub dll.** Эта библиотека должна быть установлена и зарегистрирована как на серверном, так и на клиентском компьютере. Если сервер рассчитан на стандартный маршалинг, его разработчик должен также создать **proxy/stub dll** и распространять её вместе со своим сервером. Некоторые среды разработки позволяют автоматизировать построение **proxy/stub dll** на основе формального описания интерфейсов. Delphi в число таких средств, к сожалению, не входит. Тем не менее, на Delphi можно разработать сервер, ориентированный на стандартный маршалинг, хотя **proxy/stub dll** придётся создавать другими средствами. При **стандартном маршалинге** разработчик имеет большой выбор типов для параметров и может определять свои структуры.

2. **Универсальный маршалинг**. В этом случае заместитель и заглушка реализуются стандартной системной библиотекой **oleaut32.dll**. Для того, чтобы эта библиотека могла учесть особенности конкретных интерфейсов, на клиентском и серверном компьютерах должна быть зарегистрирована библиотека типов, которую предоставляет разработчик сервера. Библиотека типов (type library) обычно представляет собой файл с расширением **tlb**, хранящий описание интерфейсов. Библиотека типов может быть также включена в сам сервер (как во внутренний, так и во внешний), и тогда сервер становится самодостаточным, никаких дополнительных файлов для работы с ним не требуется. При

использовании **универсального маршалинга** разрешёно использовать только **VARIANT-совместимые типы**.

3. **Пользовательский маршалинг**. В этом случае объект сам занимается упаковкой данных, реализуя стандартный интерфейс **IMarshal**. Разработчик сервера при этом должен предоставить оригинальную библиотеку для создания заместителя в адресном пространстве клиента, и этот заместитель также должен реализовывать интерфейс **IMarshal**. В этом случае разработчик сервера получает возможность не только управлять процессами прямого и обратного маршалинга, но и выбрать свой способ передачи данных, отказавшись от услуг стандартных объектов канала. Это позволяет, например, использовать нестандартный транспортный протокол или даже нестандартное физическое соединение при взаимодействии двух компьютеров.

Следует отметить, что существует некоторая неоднозначность в понятии **"стандартный маршалинг".** Дело в том, что большинство стандартных интерфейсов маршалируются через **oleaut32.dll**, поэтому данная библиотека обычно называется стандартным маршалером. Соответственно, маршалинг с помощью этой библиотеки также иногда называют стандартным, создавая путаницу с маршалингом через оригинальную **proxy/stub dll**. В частности, в справке Delphi маршалинг через библиотеку типов и **oleaut32.dll** называется стандартным (видимо, из маркетинговых соображений, потому что иначе пришлось бы писать, что Delphi не поддерживает в полном объёме стандартный маршалинг, а это звучит не очень хорошо: стандартный — и вдруг не поддерживает). Мы здесь под **стандартным маршалингом** всегда будем подразумевать маршалинг через **proxy/stub dll**.

При регистрации интерфейса в реестре указывается способ его маршалинга и **proxy/stub dll** или **библиотека типов**, которые будут при этом использоваться (исключение составляет пользовательский маршалинг — при его использовании интерфейс вообще не нуждается в регистрации в реестре). Тип маршалинга является неотъемлемой частью интерфейса, т.е. во всех ситуациях для маршалинга каждого конкретного интерфейса используется один и тот же метод и одна и та же библиотека типов или proxy/stub dll.

Ранее мы говорили, что заместитель можно упрощённо рассматривать как COM-объект, импортирующий все те же интерфейсы, что и удалённый объект, который он замещает. Этот объект называется **менеджером заместителей (proxy manager).** Он управляет заместителями интерфейсов. **Отдельный заместитель** — это **COM-объект**, реализующий два интерфейса: **IRpcProxyBuffer** и тот интерфейс, для маршалинга которого используется данный заместитель (в некоторых случаях один заместитель интерфейса может отвечать за маршалинг сразу

нескольких интерфейсов — в этом случае он реализует их все плюс **IRpcProxyBuffer**). Менеджер заместителей самостоятельно реализует интерфейсы **IUnknown** и **IMarshal**. **IRpcProxyBuffer** используется только для взаимодействия между менеджером заместителей и заместителями интерфейсов, клиент не получает доступ к этим интерфейсам.

Таким образом, клиент "видит" в своём адресном пространстве COM-объект, который реализует все интерфейсы требуемого ему COM-объекта (и, кроме них, ещё **IMarshal, который используется только системой, но не самим клиентом**). Этот COM-объект сам по себе состоит из компонентов, которые в общем случае реализуются разными библиотеками. Действительно, COM-объект сервера может реализовывать разные интерфейсы, которые маршалируются разными способами или библиотеками. Например, **интерфейс IUnknown всегда маршалируется библиотекой oleaut32.dll** независимо от того, как маршалируются остальные интерфейсы объекта. Соответственно, может понадобиться загрузить в адресное пространство клиента несколько разных библиотек, каждая из которых создаст своего заместителя для маршалинга разных интерфейсов одного объекта. Тем не менее, эта сложная структура скрыта от клиента, ему заместитель представляется единым объектом.

Объект канала — это тоже **COM-объект,** он реализует интерфейс **IRpcChannelBuffer**. Каждый заместитель интерфейса получает указатель на **IRpcChannelBuffer**, через который он взаимодействует с объектом канала. По требованию заместителя объект канала создаёт буфер, который заместитель затем заполняет данными, и отправляет его объекту канала на стороне сервера.

Заглушка интерфейса также представляет собой **COM-объект**. Она реализует интерфейс **IRpcStubBuffer**. Объект канала серверной стороны, получив вызов, использует этот интерфейс для передачи заглушке полученных данных. Заглушки разных интерфейсов работают независимо друг от друга, поэтому никакого менеджера заглушек, объединяющего их, нет. Но в документации COM/DCOM рекомендуется думать, что такой менеджер как бы существует концептуально, потому что реально разрозненные заглушки имитируют единый клиент.

Схему передачи вызовов через границы адресного пространства с помощью заглушек и заместителей иллюстрирует рисунок 5.1. В нём предполагается, что сервер реализует интерфейсы **IInterface1** и **IInterface2**, каждый из которых имеет собственного заместителя. Использованное на рисунке обозначение интерфейсов в виде отрезков с окружностями на концах является общепринятым и рекомендуется к использованию во всех графических изображениях COM-объектов.

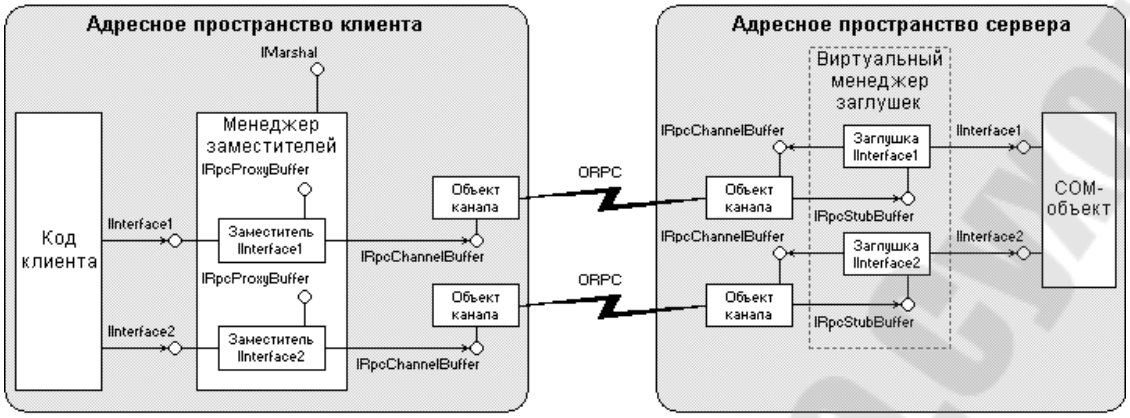

Рисунок 5.1

Когда клиент пытается получить указатель на интерфейс внешнего COM-объекта в первый раз, система запрашивает у COM-объекта сервера интерфейс **IMarshal** (этот запрос выполняется обычным образом, через **IUnknown.QueryInterface**). Если такой интерфейс у COM-объекта имеется, система через его функции получает информацию о том, какой COM-объект будет выполнять маршалинг на стороне клиента и загружает соответствующую **dll** в адресное пространство клиента. Если интерфейс **IMarshal** у COM-объекта отсутствует, или для выбранного интерфейса COM-объект не реализует пользовательский маршалинг, система читает информацию о запрошенном интерфейсе из реестра. Если там указано, что следует использовать **стандартный маршалинг**, соответствующая **proxy/stub dll** загружается в адресное пространство клиента и сервера. Если используется **универсальный маршалинг**, из реестра извлекается информация о местонахождении библиотеки типов, и в адресное пространство клиента и сервера загружается **oleaut32.dll**, которая строит заглушку и заместитель на основании данных из этой библиотеки. **Для клиента этот процесс совершенно прозрачен: система выполняет все эти действия без его участия**. Сервер, за исключением случаев пользовательского маршалинга, так же сам по себе не участвует в процессе создания заместителя и заглушки, но он должен сопровождаться либо библиотекой типов, либо proxy/stub dll, чтобы система могла выполнить вышеперечисленные действия.

Следует заметить, что маршалинг иногда требуется и при использовании внутренних COM-серверов, когда клиент и сервер находятся в одном адресном пространстве. COM-сервер, реализованный в виде dll, также должен сопровождаться библиотекой типов, proxy/stub dll или реализовывать IMarshal, иначе область его применения будет сильно ограничена.

Единственный тип сервера, создание которого Delphi поддерживает в полном объёме — это сервер, использующий универсальный маршалинг, с библиотекой типов, интегрированной в него. Для создания серверов другого типа могут потребоваться сторонние средства и/или вмешательство в работу стандартных компонентов **TComServer**, **TComObject** и т.п., а то и вовсе отказ от них. Так что для выбора другого типа сервера при программировании на Delphi должны быть очень серьёзные причины.

6 Технология OLE Automation или автоматизация

Стандарт COM основан на едином для всех поддерживающих его языков формате таблицы, описывающей ссылки на методы объекта, реализующего интерфейс. Однако вызов методов при помощи этой таблицы доступен только для компилирующих языков программирования. В то же время очень удобно было бы иметь доступ ко всем предоставляемым COM возможностям из интерпретирующих языков, таких как **VBScript**. Для поддержки этих языков была разработана технология под названием **OLE Automation** (автоматизация OLE, или просто автоматизация), позволяющая приложениям делать свою функциональность доступной для гораздо большего числа клиентов.

Основным преимуществом **автоматизации** является независимость от языка, т.е. контроллеры автоматизации могут управлять сервером независимо от языка, на котором был написан сервер (или клиент). Ваши программы могут быть "незаконченными", т.е. в них будет "заложена" возможность дополнения, расширения или даже изменения логики, интерфейса, возможностей. Таким образом, после незначительной работы другого программиста по программированию вашей программы (программированию в тех рамках, в которых вы это предполагали), ваш программный продукт сможет решать задачи, решения которых изначально не предполагалось. Кроме того, стоит отметить, что автоматизация поддерживается на уровне операционной системы (Windows).

6.1 Основные термины

Автоматизация (automation или OLE-automation) - технология, позволяющая приложениям и библиотекам (DLL) предоставлять свои программируемые объекты с целью их использования в других приложениях.

Сервера автоматизации (automation servers) (СА) - приложения и библиотеки (DLL), предоставляющие свои программируемые объекты.

Контроллеры автоматизации (automation controllers) (КА) - приложения, получающие доступ к управлению программируемыми объектами, предоставляемыми серверами автоматизации.

Таким образом, из вышеприведенных определений становится вполне понятным, что контроллеры автоматизации **(КА)** способны "программировать" сервер автоматизации **(СА)** с помощью некоторого макроязыка, предлагаемого **СА**.

Объекты автоматизации (ОА) - это обычные **COM-объекты**, в которых помимо интерфейса **IUnknown** реализован интерфейс **IDispatch**.

Технология **OLE Automation** базируется на технологии COM и является ее подмножеством, однако накладывает на COM-серверы ряд дополнительных требований:

- интерфейс, реализуемый COM-сервером, должен наследоваться от интерфейса **IDispatch**;
- должны использоваться типы данных из числа поддерживаемых технологией **OLE Automation** (таблица 3);
- все методы интерфейса должны быть либо процедурами, поддерживающими соглашение о вызове **safecall**, либо функциями, поддерживающими соглашение о вызове **safecall** и возвращающими значение типа HRESULT;
- для поддержки пользовательских типов данных должен быть реализован интерфейс **IRecordlnfo**.

Таблица 3 - Типы данных OLE Automation

Тип (Delphi)	Тип (IDL)	Описание
Byte	Byte	1 байт, целое без знака, диапазон от 0 до 255
Currency	CURRENCY	8 байт, с плавающей запятой и 4 знаками после запятой, диапазон от -922 337 203 685 477,5808 до 922 337 203 685 477,5807, сопроцессорный тип
DISPPARAMS	DISPPARAMS	Структура, содержащая параметры вызова методов через метод **Invoke** интерфейса **IDispatch**. Расшифровка структуры приведена в модуле **ActiveX.pas**
Double	Double	8 байт, с плавающей запятой, диапазон от 5,0x10-324 до 1,7x10308, 15-16 знаков
EXCEPINFO	EXEPINFO	Структура, содержащая информацию об исключении. Расшифровка приведена в модуле ActiveX.pas
GUID	GUID	Глобально уникальный идентификатор (класса, интерфейса). Структура размером 16 байт
Hresult	HRESULT	4 байта, целое число без знака, диапазон от 0 до 4 294 967 295
Integer	Long	4 байта, целое число со знаком, диапазон от 2 147 483 648 до 2 147 483 647

Largeuint	unsigned int64	8 байт, целое число со знаком, диапазон от 0 до 2^{64}-1
OleVariant	VARIANT	Содержит любые данные, тип может меняться динамически. Минимальный размер — 16 байт
Pchar	LPSTR	Указатель на строку, 4 байта
PwideChar	LPWSTR	Указатель па строку, в который для храпения каждого символа используют 2 байта. Размер — 4 байта
PSafeArray	SAFEARRAY	Указатель на массив целых чисел, 4 байта
Shortint		1 байт, целое со знаком, диапазон от-128 до 127
Single	Float	4 байта, с плавающей запятой, диапазон от $1{,}5\cdot10^{45}$ до $3{,}4\cdot10^{38}$, 7-8 знаков
SmallInt	Short	2 байта, целое со знаком, диапазон от - 32 768 до 32 767
SYSINT	Int	Системная целая переменная со знаком, в 32-разрядных операционных системах совпадает с типом integer
SYSUINT	unsigned int	Системная целая переменная без знака, в 32-разрядных операционных системах совпадает с типом 1ongword. Другое название переменной этого типа — Cardinal
TdateTime	DATE	8 байт, с плавающей запятой, целая часть — число дней, прошедших с 30 декабря 1899 года, дробная часть — доля от 24 часов
Tdecimal	DECIMAL	Структура, содержит число с плавающей запятой и точность его представления (количество значимых десятичных знаков). Расшифровка содержится в модуле ActiveX.pas
WideString	BSTR	Строка переменной длины, для храпения каждого символа используется 2 байта
Word	unsigned short	2 байта, целое без знака, диапазон от 0 до 65 535
WordBool	VARIANT_B00L	2 байта, логическая переменная (True = -1, False = 0)
Int64	_int64	8-байтовое целое число
FONTBOLD,	FONTBOLD,	Параметры шрифта

FONTITALIC, FONTNAME, FONTSIZE, FONTSTRIKE THROUGH	FONTITALIC, FONTNAME, FONTSIZE, FONTSTRIKET HROUGH	
SCODE	SCODE	32-битовос значение статуса, используемое в MAPI
Longword	unsigned int	4 байта, целое число без знака, диапазон от 0 до 4 294 967 295

6.2 Интерфейс IDispatch

В технологии COM предусмотрена возможность доступа к методам интерфейса при отсутствии информации о порядке реализации методов в виртуальной таблице. При реализации интерфейсов среда разработки может сохранить текстовые названия методов в файле, в котором находится скомпилированный код COM-сервера. Это означает, что в COM-сервере был реализован интерфейс **IDispatch**, который является центральным элементом технологии OLE Automation.

Интерфейс **IDispatch** определён в модуле **System** следующим образом:

```
type
  IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult;
stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID:
Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr:
Pointer): HResult; stdcall;
  end;
```

Ключевыми методами этого интерфейса являются методы **GetIDsOfNames** и **Invoke**, которые позволяют клиенту получить у сервера ответ па вопрос, поддерживает ли он метод с указанным именем, а затем, если метод поддерживается, — вызвать его.

Когда клиенту требуется вызвать какой-либо метод сервера, сначала он вызывает метод **GetIDsOfNames** интерфейса **IDispatch**, передавая ему имя запрошенного метода сервера. Если сервер поддерживает запрошенный метод, он возвращает его идентификатор — целое число, уникальное для каждого метода. После этого клиент упаковывает параметры в массив переменных типа **OleVariant** и вызывает метод **Invoke**, передавая ему массив параметров и идентификатор запрошенного метода.

Метод **Invoke** имеет следующие параметры и свойства:

DispID - содержит диспетчерский идентификатор (dispatch ID), это число, которое показывает какой метод должен быть вызван в сервере.

IID - на данный момент не используется.

LocaleID - содержит информацию о языке.

Flags - определяет какого типа метод будет вызван в сервере: метод доступа к свойствам или обычный метод.

Params - содержит указатель на массив **TDispParams**, который содержит параметры, передаваемые этому методу.

VarResult - это указатель на переменную типа **OleVariant**, в которую будет записано возвращаемое значение вызываемого метода.

ExcepInfo - указатель на запись типа **TExceptInfo**, которая будет содержать информацию об ошибке, в случае, если метод **Invoke** возвращает значение **DISP_E_TYPEMISMATCH** или **DISP_E_PARAMNOTFOUND**.

ArgErr - указатель на целое число (индекс некорректного параметра в массиве **Params**).

Таким образом, все, что должен знать клиент, — это строковое имя метода. Такой алгоритм позволяет работать с наследниками интерфейса **IDispatch** из языков сценариев.

Методы **GetTypeInfo** и **GetTypeInfoCount** являются вспомогательными и обеспечивают поддержку библиотеки типов объекта. Реализация методов **GetIDsOfNames** и **Invoke**, предоставляемая COM по умолчанию, базируется на библиотеке типов объекта.

При работе с интерфейсом **IDispatch** связь между вызываемыми методами устанавливается при выполнении приложения. Такой способ вызова методов интерфейсов называют **поздним связыванием** (**late binding**). Для подобного вызова методов требуется провести большее число операций, чем при прямом обращении к виртуальной таблице, и это может сказаться на скорости выполнения вызовов. При отсутствии библиотеки типов на этапе разработки сервера невозможно проверить правильность написания имен методов и списков параметров методов. Это может обнаружиться только на этапе выполнения клиентского приложения — когда произойдет исключение. Поэтому клиентское приложение,

работающее с сервером через интерфейс **IDispatch**, обязательно следует тестировать на предмет корректного выполнения всех команд.

Кроме того, при наличии библиотеки типов для реализации нотификационных сообщений (то есть уведомлений о событиях, которые COM-сервер посылает клиенту) в элементах управления **ActiveX** вывод этих сообщений возможен только через интерфейс **IDispatch**. Элемент управления **ActiveX** определяет интерфейс для поддержки нотификационных сообщений, но сам его не создает. Он должен быть реализован в клиенте, а элемент управления **ActiveX** должен получить ссылку на него и вызывать методы этого интерфейса в ответ на события, происходящие с ним. Поскольку COM-сервер следует компилировать раньше клиента, нет никакой возможности на этапе компиляции сервера получить доступ к таблице виртуальных методов клиента и связать нотификационные сообщения с методами, определенными в клиенте.

6.2 Поддержка автоматизации в Delphi

Delphi обеспечивает поддержку клиентов автоматизации. Тип данных **Variant** может содержать ссылку на интерфейс **IDispatch** и использоваться для вызова его методов.

Рассмотрим простой пример использования сервера автоматизации Internet Explorer. Ниже приведен код модуля контроллера автоматизации.

```
unit Test_CA;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs,ComObj, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var V: Variant;
begin
V:=CreateOleObject('InternetExplorer.Application');
V.Visible:=TRUE;
V.Toolbar :=FALSE;
V.Left:=(Screen.Width-600) div 2;
V.Width:=600;
V.Top:=(Screen.Height-400) div 2;
V.Height:=400;
V.Navigate(URL:='file:///D:/temp/12345.html');
V.StatusText:=V.LocationURL;
Sleep(10000);
V.Quit;
end;

end.
```

Приведенный выше код весьма необычен и заслуживает внимательного рассмотрения:

- очевидно, что переменная V не обладает ни одним из используемых свойств и методов;
- вызываемые свойства и методы нигде не описаны, однако это не ведет к ошибке компиляции;
- объект создается не по **CLSID**, а по информативному имени функцией **CreateOleObject**.

Все это непривычно и выглядит довольно странно. На самом деле — ничего странного нет. **Компилятор Delphi** просто запоминает в коде программы строковые описания обращений к серверу автоматизации, а на этапе выполнения передает их интерфейсу **IDispatch** сервера, который и производит синтаксический анализ и выполнение.

Если исправить третью строку процедуры следующим образом:

```
V.Leftl := (Screen.Width - 600) div 2;
```

программа успешно откомпилируется, однако при попытке выполнения выдаст ошибку с сообщением, что метод **Leftl** не поддерживается сервером автоматизации. Такое обращение к серверу, как уже упоминалось, называется поздним связыванием, означающим, что связывание имен свойств и методов объекта с их кодом происходит не на этапе компиляции, а на этапе выполнения программы.

Достоинства позднего связывания очевидны — не нужна библиотека типов, написание несложных программ упрощается. Столь же очевидны недостатки — не производится контроль вызовов и передаваемых параметров па этапе компиляции, работает приложение несколько медленнее, чем при раннем связывании.

Если COM-сервер находится в другом апартаменте, временные затраты па позднее связывание пренебрежимо малы по сравнению с затратами на маршалинг вызовов. Разница в скорости между ранним и поздним связыванием становится ощутимой (десятки и сотни раз) при нахождении клиента и сервера в одном апартаменте, что возможно только для внутрипроцессного сервера при совместимой с клиентом модели потоков. Для внепроцессного сервера (сервера, размещенного в отдельном исполняемом файле) затраты на вызов метода путем раннего и позднего связывания практически равны.

Главным преимуществом раннего связывания является строгий контроль типов на этапе компиляции. Для разрешения проблемы нестрогого контроля типов COM предлагает несколько дополнительных возможностей.

6.3 Диспинтерфейс

Диспинтерфейс (dispinterface) — это декларация методов, доступных через интерфейс **IDispatch**. Объявляется диспинтерфейс следующим образом:

```
type
    IMyDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}' ]
    property Count: Integer dispid 1
    procedure Clear dispid 2;
end:
```

Самих методов может физически и не существовать (например, они реализуются динамически в методе **Invoke**).

Рассмотрим использование диспинтерфейса па простом примере. Объявим диспиптерфейс объекта InternetExplorer и задействуем его в своей программе:

```
type
IIE = dispinterface
['{0002DF05-0000-0000-C000-000000000046}']
property Visible: WordBool dispid 402;
end;

procedure TForml.ButtonlClick(Sender: TObject):
var
IE: IIE;
begin
IE := CreateOleObject('InternetExplorer.Application') as IIE;
IE.Visible := True;
End;
```

Эта программа успешно компилируется и работает, несмотря на то, что в интерфейсе объявлено только одно свойство из множества имеющихся свойств и методов. Это происходит благодаря тому, что **Delphi** не вызывает методы **диспинтерфейса** напрямую и поэтому не требует полного описания всех методов в правильном порядке. При вызове метода диспинтерфейса **Delphi** просто вызывает метод **Invoke** соответствующего метода **IDispatch**, передавая ему идентификатор метода, указанный в параметре **dispid**. В результате программист получает возможность строго контролировать типы при вызове методов интерфейса **IDispatch** и вызывать методы, описанные в **диспинтерфейсе**, без формирования сложных структур данных, требующихся для вызова метода **Invoke**. Необходимо лишь указать (или импортировать из библиотеки типов сервера) описание диспинтерфейса.

6.4 Дуальные интерфейсы

Идея дуальных интерфейсов (**dual interfaces**) очень проста. Сервер реализует одновременно некоторый интерфейс **VTable**, оформленный по стандартам COM, и **диспинтерфейс**, доступный через интерфейс **IDispatch**. При этом интерфейс **VTable** должен наследоваться от **IDispatch** и иметь идентичный с **диспинтерфейсом** набор методов. Такое оформление сервера позволяет клиентам работать с ним наиболее удобным для каждого клиента образом.

Клиенты, применяющие интерфейс **VTable**, вызывают его методы напрямую, а клиенты, использующие позднее связывание, — через методы интерфейса **IDispatch**.

Примечание:Большинство OLE-серверов реализуют дуальный интерфейс.

Если объекты автоматизации разрабатываются с помощью встроенного в Delphi **Automation Object Wizard** (**AOWizard**), то **OA** автоматически поддерживают двойной интерфейс, т.е. методы **OA** можно вызывать как с помощью метода **Invoke,** так и с помощью потомков интерфейса **IDispatch**.

6.5 Создание внешнего сервера автоматизации и контроллера для работы с ним

Внешний сервер автоматизации (LocalServer) - является выполняемым файлом, который может создавать **OA** для использования их другими приложениями. Из названия понятно, что внешние **CA** выполняются в контексте своего собственного процесса. Как и все COM-объекты **CA** должны быть зарегистрированы, т.е. должны создавать такие же записи в реестре, как и все остальные COM-объекты, плюс два дополнительных параметра.

Если **CA** реализуется в Delphi, то регистрация происходит автоматически при первом запуске **CA** при выполнении метода **Application.Initialize.**

Рассмотрим создание сервера автоматизации и контроллера на примере. Допустим требуется разработать сервер автоматизации, на основе приложения (текстовый редактор), которое содержит панель инструментов с четырьмя кнопками и компонент TMemo, а также диалоговые окна открытия и сохранения файла.

Последовательность действий для решения поставленной задачи:

1. ***Создание Текстового редактора***. Создадим обычное приложение (рисунок 6.1)

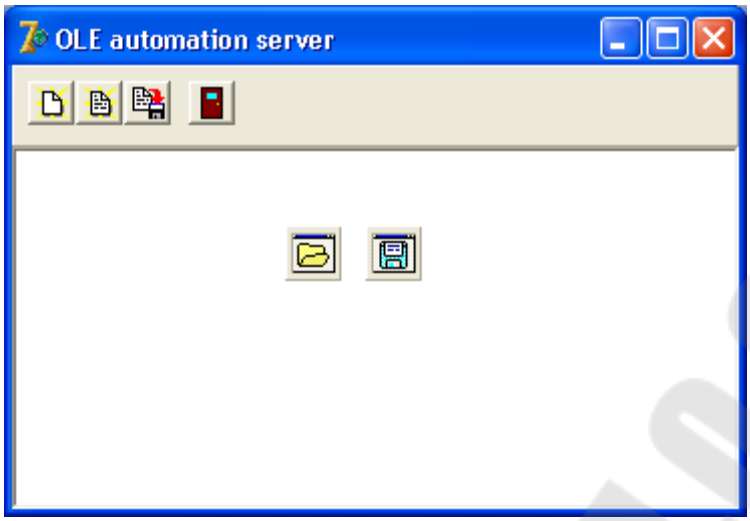

Рисунок 6.1

В тексте модуля напишем коды обработчиков событий, связанные со щелчками на кнопках

```
unit UServ1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, Buttons, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Memo1: TMemo;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    procedure SpeedButton1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure SpeedButton3Click(Sender: TObject);
  private
    { Private declarations }
```

```
public
   { Public declarations }
end;

var
  Form1: TForm1;

implementation
{$R *.dfm}

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  Memo1.Lines.Clear;
end;

procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
end;

procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  if SaveDialog1.Execute then
Memo1.Lines.SaveToFile(SaveDialog1.FileName);
end;

procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  Close;
end;

end.
```

Сохраним проект под именем AutoServer

2. ***Создание Сервера автоматизации***. Отметим, что созданный нами текстовый редактор представляет собой обычное Windows-приложение и не является сервером автоматизации.

Для создания ***Сервера автоматизации*** добавим объект **Automation Object** на странице **ActiveX** репозитария объектов (рисунок 6.2)

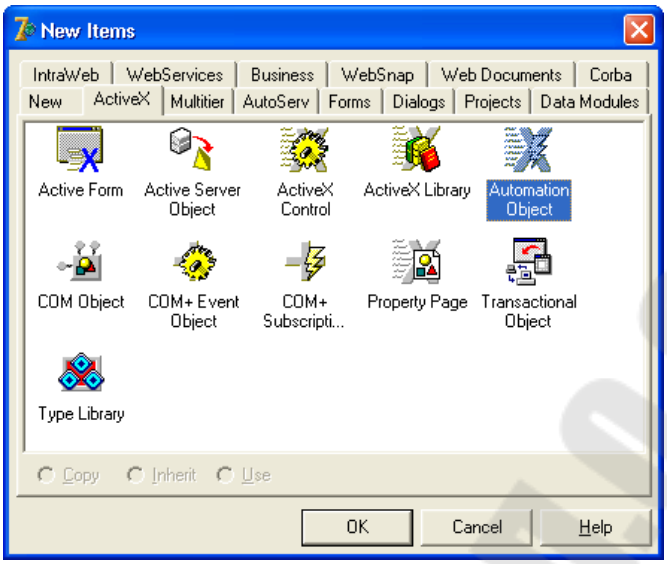

Рисунок 6.2

В открывшемся диалоговом (рисунок 6.3) окне введем имя, под которым данный класс COM-объектов будет зарегистрирован в реестре. В раскрывающемся списке **Threading Model** выберем пункт **Free**, означающий выбор модели свободных потоков **(free-threaded model).** В раскрывающемся списке **Instancing** выбираем **Multiple Instance.**

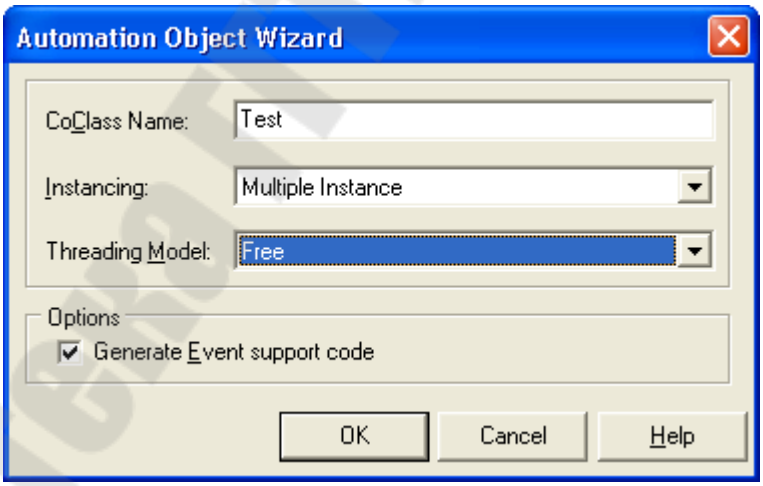

Рисунок 6.3

Флаг **Generate Event support code** требуется для поддержки нотификационных сообщений в сервере автоматизации. После щелчка на кнопке OK появится заготовка модуля с рядом определенных переменных и методов, связанных с поддержкой нотификационных сообщений. Если флаг **Generate Event support code** не установлен, то методы, поддерживающие нотификационные сообщения, не генерируются, кроме

этого в библиотеке типов будет отсутствовать описание интерфейса **ITestEvents**.

В редакторе библиотеки типов (рисунок 6.4) определяем свойства и методы созданного класса COM-объектов.

Для нашего сервера опишем методы **NewFile**, **OpenFile**, **SaveFile**, **AddLine** и их параметры, а также свойства **Text**, **Width** и **Visible**.

Метод **FileNew** параметров не имеет. Методы **OpenFile** и **SaveFile** имеют один строковый параметр типа **BSTR (WideString)** - имя файла. Метод **AddLine** также имеет один строковый параметр, задающий добавляемую строку. Свойство **Text** доступно для **чтения** и **записи** и имеет тип **BSTR (WideString).** Свойство **Visible** имеет логический тип **VARIANT_BOOL (WordBool)** и тоже доступно для **чтения** и **записи**. Свойство **Width** имеет целый тип **long (integer),** определяет число пикселов и также доступно как для чтения, так и для записи.

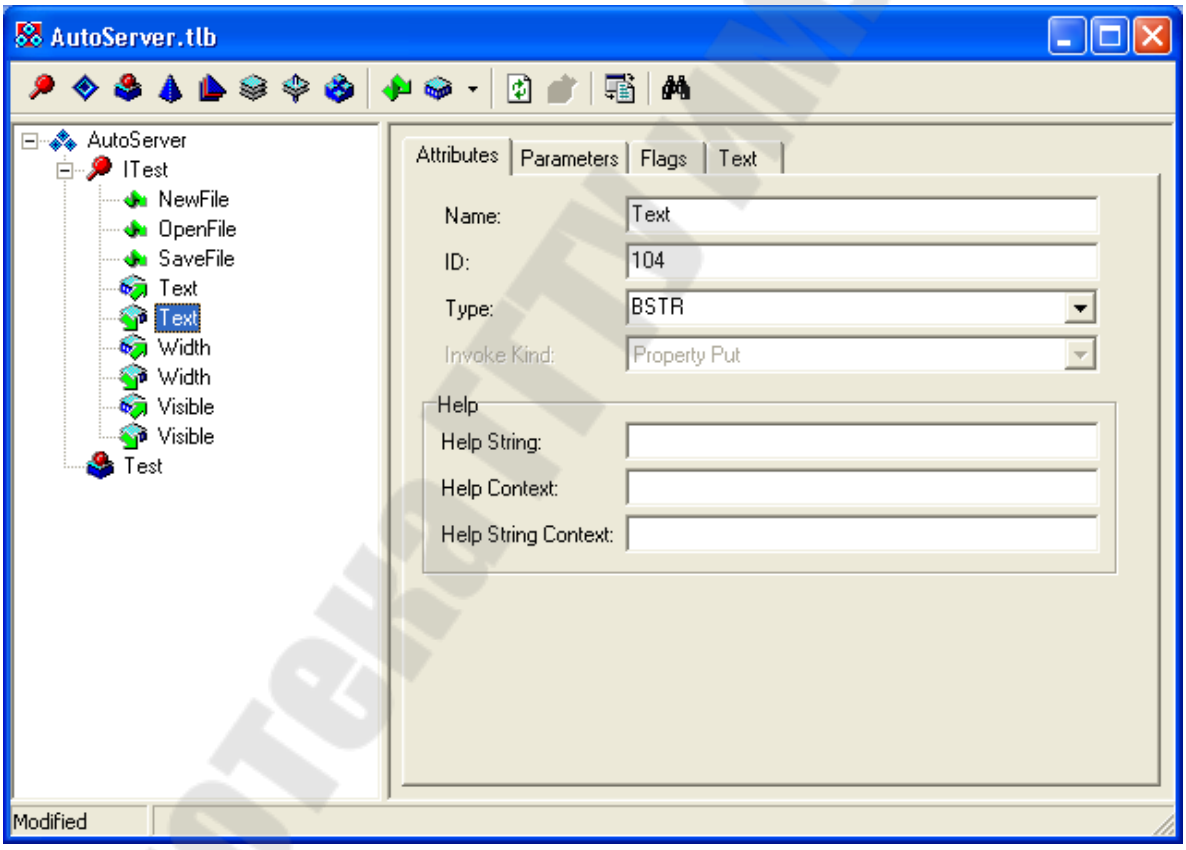

Рисунок 6.4

3. ***Реализация методов сервера автоматизации***. Перейдем к реализации методов. Для этой цели в окне редактора библиотеки типов следует щелкнуть на кнопке **Refresh** панели инструментов. В модуль реализации (сгенерированный ранее мастером, использованным при

создании сервера) будут добавлены заготовки соответствующих методов. Добавим код реализации этих методов.

```
function TTest.Get_Text: WideString;
begin
Result := Form1.Memo1.Text;
end;

function TTest.Get_Visible: WordBool;
Result:= Form1.Visible;
end;

function TTest.Get_Width: Integer;
begin
Result := Form1.Width;
end;

procedure TTest.AddLine(const S: WideString);
begin
Form1.Memo1.Lines.Add(S);
end;

procedure TTest.NewFile;
begin
Form1.SpeedButton1Click(nil);
end;

procedure TTest.OpenFile(const S: WideString);
begin
if Length(S)>0 then
Form1.Memo1.Lines.LoadFromFile(S)
else
Form1.SpeedButton2Click(nil);
end;

procedure TTest.SaveFile(const S: WideString);
begin
if Length(S) > 0 then
Form1.Memo1.Lines.SaveToFile(S)
else
Form1.SpeedButton3Click(nil);
end;
```

```
procedure TTest.Set_Text(const Value: WideString);
Form1.Memo1.Text := Value;
end;

procedure TTest.Set_Visible(Value: WordBool);
begin
Form1.Visible := Value;
end;

procedure TTest.Set_Width(Value: Integer);
begin
Form1.Width:=Value;
end;
```

4. ***Регистрация сервера автоматизации***. Регистрация сервера автоматизации происходит автоматически после запуска на выполнение.

5. ***Создание контроллера автоматизации***. Создаем новое приложение (рисунок 6.5).

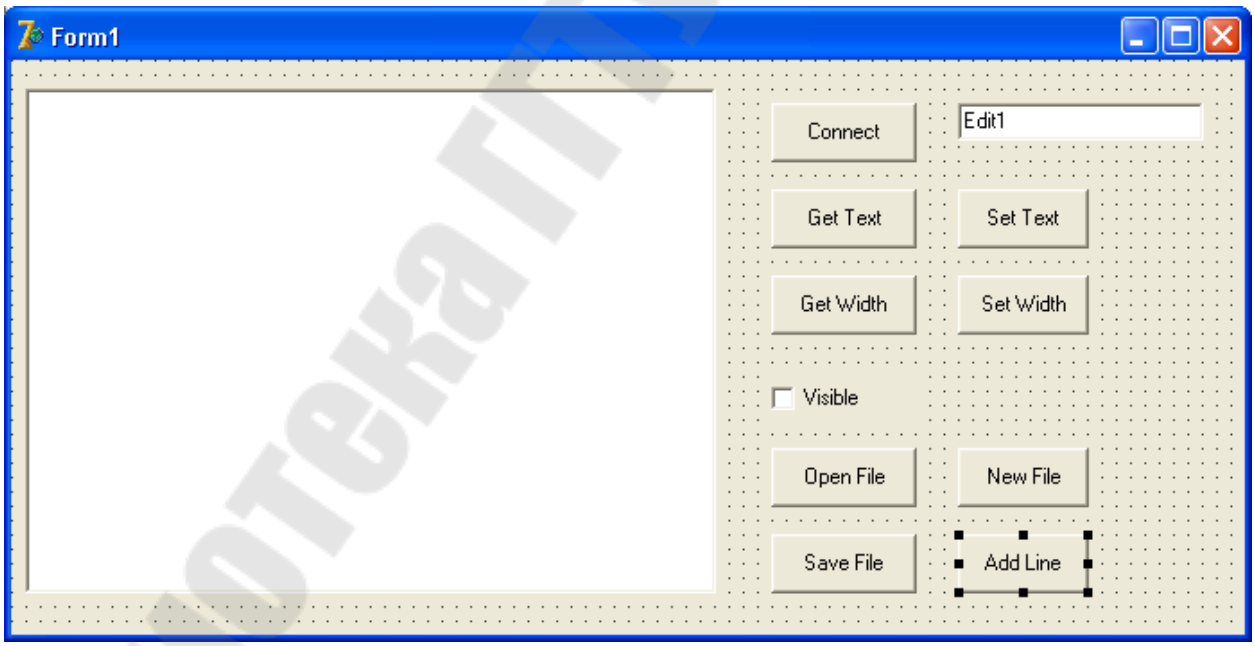

Рисунок 6.5

В секции **uses** добавляем **ComObj** и **Variants**. В секции **private** класса **TForml** объявляем переменную **FServ** типа **Variant**. Создаем обработчики событий, связанные со щелчками на кнопках.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then begin
    FServ:=Unassigned;
    Button1.Caption:='Connect';
  end else begin
    FServ:=CreateOLEObject('AutoServer.Test');
    Button1.Caption:='Disconnect';
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  Memo1.Text:=FServ.Text;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  FServ.Text:=Memo1.Text;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  Edit1.Text:=FServ.Width;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  FServ.Width:=StrToInt(Edit1.Text);
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  FServ.Visible:=CheckBox1.Checked;
end;
```

```
procedure TForm1.Button6Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  FServ. FileOpenFile(Edit1.Text);
end;

procedure TForm1.Button7Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  FServ. FileSave (Edit1.Text);
end;

procedure TForm1.Button8Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  FServ. FileNew;
end;

procedure TForm1.Button9Click(Sender: TObject);
begin
  if vartype(FServ)=varDispatch then
  FServ.AddLine(Edit1.Text);
end;
```

Сохраняем проект под именем **ClientAuto** и запускаем на выполнение. Проверяем работоспособность сервера и контроллера автоматизации.

7 Создание контроллеров автоматизации приложений Microsoft Office

Приложения Microsoft Office наиболее часто используются не только в качестве пользовательских инструментов, но и в качестве серверов автоматизации. Многие разработчики в процессе работы над своими проектами применяют сервисы, предоставляемые Microsoft Office, такие как средства построения сводных таблиц и диаграмм с помощью Microsoft Excel, генерации и печати документов с помощью Microsoft Word и т. д. Нередко пользователи, привыкшие применять приложения Microsoft Office в повседневной работе, сами настаивают на наличии в приложениях таких сервисов или просто на сохранении отчетов и других данных в виде документов Microsoft Office.

В комплект поставки любого коммерческого сервера автоматизации обычно входит документация и справочные файлы, описывающие их объектную модель. В случае Microsoft Office 2000 — это справочные файлы для программистов на Visual Basic for Applications VBAxxx9.CHM, в случае Microsoft Office XP — файлы VBAxx10.CHM, в случае Microsoft Office 2007 — файлы VBAxx11.CHM. По умолчанию эти файлы не устанавливаются, так как нужны разработчикам, а не рядовым пользователям. Помимо этого, вся информация об объектах, нужная контроллерам автоматизации, содержится в библиотеках типов, имеющих в случае приложений Microsoft Office расширение *.olb (за исключением Excel — библиотека типов этого приложения находится в самом исполняемом файле Excel.exe). Поэтому при создании приложений, использующих раннее связывание, следует генерировать интерфейсный модуль к этим библиотекам.

7.1 Объектные модели Microsoft Office

Приложения Microsoft Office предоставляют контроллерам автоматизации доступ к своей функциональности с помощью своей объектной модели, представляющей собой иерархию объектов. Объекты могут предоставлять доступ к другим объектам посредством коллекций. В качестве иллюстрации иерархии объектов Microsoft Office приведем небольшой фрагмент объектной модели Microsoft Word (рисунок 7.1).

В объектных моделях всех приложений Microsoft Office всегда имеется самый главный объект, доступный приложению-контроллеру и представляющий самоавтоматизируемое приложение. Для всех приложений семейства Microsoft Office он носит название Application, и многие его свойства и методы также одинаковы для всех этих приложений.

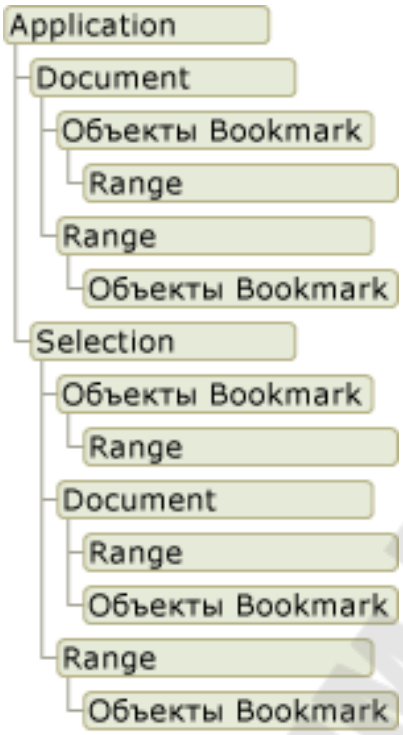

Рисунок 7.1

Ниже представлены те из них, которые используются наиболее часто:

- **Свойство Visible** (доступное для объекта Application всех приложений Microsoft Office) позволяет приложению появиться на экране и быть представленным в панели задач; оно принимает значения True (пользовательский интерфейс приложения доступен) или False (пользовательский интерфейс приложения недоступен; это значение устанавливается по умолчанию).Если вам нужно сделать что-то с документом Office в фоновом режиме, не информируя об этом пользователя, можно не обращаться к этому свойству — в этом случае приложение можно будет найти только в списке процессов с помощью программы Task Manager.
- **Метод Quit** закрывает приложение Office. В зависимости от того, какое приложение Office автоматизируется, он может иметь или не иметь параметры.

7.2 Общие принципы создания контроллеров автоматизации Microsoft Office

В общем случае контроллер автоматизации должен выполнять следующие действия:

1. Проверить, запущена ли копия приложения-сервера.

2. В зависимости от результатов проверки (либо назначения данного контроллера) запустить копию автоматизируемого приложения Office или подключиться к уже имеющейся копии.

3. Если необходимо, сделать окно приложения-сервера видимым.

4. Выполнить какие-то действия с приложением-сервером (например, создать или открыть документы, изменить их данные, сохранить документы и др.).

5. Закрыть приложение-сервер, если его копия была запущена данным контроллером, или отключиться от него, если контроллер подключился к уже имеющейся копии.

Соответствующий код-заготовка для Delphi представлен ниже:

```
uses ComObj, ActiveX
…….
procedure TForm1.Button1Click(Sender: TObject);
var
ServerIsRunning : Boolean;
Unknown :         IUnknown;
Result :                HResult;
AppProgID :             String;
App :             Variant;
begin
// Указать программный идентификатор приложения-сервера
AppProgID := 'Word.Application';
ServerIsRunning := False;
Result := GetActiveObject(ProgIDToClassID(AppProgID),nil,Unknown);
if (Result = MK_E_UNAVAILABLE) then
// Создать один экземпляр сервера
App := CreateOleObject(AppProgID)
else begin
// Соединиться с уже запущенной копией сервера
App := GetActiveOleObject(AppProglD);
ServerIsRunning := True;
end;
// показать окно приложения на экране
App.Visible := True;
//……..
// Здесь выполняются другие действия
// с объектами приложения Office
//……..
if not ServerIsRunning then App.Quit;
```

```
App:=Unassigned;
end;
```

В данном коде мы воспользовались функциями **GetActiveOleObject** и **CreateOleObject** для подключения к уже запущенной копии приложения-сервера или запуска новой, если сервер не запущен, что приводит к тому, что в вариантную переменную помещается ссылка на объект Application соответствующего сервера.

Сделаем одно маленькое замечание касательно числа параметров методов объектов автоматизации. В случае позднего связывания число указанных в коде параметров метода не обязано совпадать с их истинным числом, которое можно найти в описании объектной модели соответствующего приложения. В этом случае, несмотря на то, что метод Quit объекта Application для некоторых приложений Microsoft Office (например, Microsoft Word) имеет параметры, вполне допустимым является следующий код:

```
App.Quit;
```

С другой стороны, при раннем связывании следует строже подходить к определению параметров — их число и типы должны соответствовать описанию методов в библиотеке типов. Например, в случае раннего связывания корректный код на Delphi для закрытия документа Word со значениями всех параметров по умолчанию будет иметь вид:

```
App.Quit(EmptyParam, EmptyParam, EmptyParam);
```

Кроме этого, как правило, при создании контроллеров автоматизации подобных приложений не рекомендуется предоставлять конечному пользователю доступ к пользовательскому интерфейсу сервера, по крайней мере, в те моменты, когда свои действия выполняет контроллер. В противном случае из-за возможного нежелательного и непредсказуемого вмешательства пользователя результат работы контроллера может оказаться отличным от ожидаемого. Одним из средств решения данной проблемы является манипуляция свойством Visible объекта Application — пока это свойство равно False, окна приложения не только невидимы, но и неспособны обрабатывать события, инициируемые мышью или клавиатурой, и поэтому вмешательство конечного пользователя в их работу исключено.

7.3 Автоматизация Microsoft Word

Рассмотрим программные идентификаторы основных объектов Microsoft Word и объектную модель этого приложения.

Объекты, непосредственно доступные приложению-контроллеру, представлены в следующей таблице:

Таблица 7.1

Объект	**Программный идентификатор**	**Комментарий**
Application	Word.Application	С помощью этого программного идентификатора создастся экземпляр Word без открытых документов
	Word.Application.9(10)	
Document	Word.Document	С помощью этого программного идентификатора создается экземпляр Word с одним вновь созданным документом
	Word.Document.9(10)	
	Word.Template.8	

Все остальные объекты Word являются так называемыми внутренними (internal) объектами. Это означает, что они не могут быть созданы сами по себе. Например, объект Paragraph (абзац) не может быть создан отдельно от содержащего его документа.

Основным объектом, как и в объектных моделях других приложений Microsoft Office, является объект Application, содержащий коллекцию Documents объектов типа Document.

Каждый объект типа Document содержит коллекцию Paragraphs объектов типа Paragraph, коллекцию Bookmarks объектов типа Bookmark, коллекцию Characters объектов типа Character и т. д.

Манипуляция документами, абзацами, символами, закладками реально осуществляется путем обращения к свойствам и методам этих объектов.

Ниже мы рассмотрим наиболее часто встречающиеся задачи, связанные с автоматизацией Microsoft Word. При разработке примеров использования Microsoft Word можно модифицировать приведенный ранее код создания контроллера, заменив комментарии кодом, манипулирующим свойствами и методами объекта Word.Application.

7.3.1 Создание, открытие и закрытие документов Microsoft Word

Создать новый документ Word можно, используя метод Add коллекции Documents объекта Application:

App.Documents.Add;

Для того, чтобы создать нестандартный документ нужно указать имя шаблона в качестве параметра метода Add;

App.Documents.Add('C:\Program Files\Microsoft Office\ Templates\ 1033\ Manual .dot');

Для **открытия** уже существующего документа следует воспользоваться методом Open коллекции Documents:

App.Documents.Open('C:\MyWordFi1e.doc');

Свойство **ActiveDocument** объекта Word.Application указывает на текущий активный документ среди одного или нескольких открытых. Помимо этого, к документу можно обращаться по его порядковому номеру с помощью метода **Item**; например, ко второму открытому документу можно обратиться так:

App.Documents.Item(2);

Нумерация членов коллекций в Microsoft Office начинается с **единицы**.

Сделать документ активным можно с помощью метода **Activate**:

App.Documents.Item(2).Activate;

Сохранение, печать и закрытие документов Microsoft Word

Закрытие документа может быть осуществлено с помощью метода **Close** одним из
следующих способов:

App.Documents.Item(1).Close;
App.ActiveDocument.Close;

Метод **Close** имеет несколько необязательных (в случае позднего связывания) параметров, влияющих на правила сохранения документа. Первый из них определяет необходимость сохранения внесенных в документ изменений и принимает три возможных значения (соответствующие константы рекомендуется описать в приложении):

```
const
    wdDoNotSaveChanges          = $00000000; // не сохранять изменения
    wdSaveChanges               = SFFFFFFFF; // сохранять изменения
    wdPromptToSaveChanges       = $FFFFFFFE; // вывести диалоговое
окно
                                             //с соответствующим
                                                запросом
```

Второй параметр определяет формат сохраняемого документа:

```
const
    wdWordDocument              = $00000000: // сохранить в формате
Word
    wdOriginalDocumentFormat    = $00000001: // сохранить в исходном
формате
    wdPromptUser                = $00000002; // вывести диалоговое
окно Save As
```

Третий параметр принимает значения True или False и определяет необходимость пересылки документа следующему пользователю по электронной почте. Если такая функциональность не нужна, этот параметр можно проигнорировать.

Таким образом, при использовании перечисленных выше параметров закрыть документ можно, например, так:

```
App.ActiveDocument.Close(CwdSaveChanges, wdPromptUser);
```

Просто сохранить документ, не закрывая его, можно с помощью метода **Save**:

```
App.ActiveDocument.Save;
```

Этот метод также имеет несколько необязательных (в случае позднего связывания) параметров, первый из которых равен **True**, если документ сохраняется автоматически, и **False**, если нужно выводить диалоговое окно для получения подтверждения пользователя о сохранении изменений (если таковые были сделаны).

Второй параметр влияет на формат сохраняемого документа, и список его возможных значений совпадает со списком значений второго параметра метода Close.

Прмечание: При создании контроллеров автоматизации, использующих позднее связывание, можно тем не менее сгенерировать интерфейсный модуль для соответствующей библиотеки типов и сохранить его в виде текстового файла, не включая сам модуль в проект. В этом случае можно будет воспользоваться содержащимися в тексте интерфейсного модуля определениями значений констант, копируя их в код приложения-контроллера.

Закрыть само приложение Word можно с помощью метода **Quit** объекта Word.Application. Этот метод имеет в общем случае три параметра, совпадающих с параметрами метода Close объекта Document.

Вывод документа на устройство печати можно осуществить с помощью метода **Printout** объекта Document, например:

```
App.ActiveDocument.PrintOut;
```

Если нужно изменить параметры печати, следует указать значения соответствующих параметров метода Printout (их в случае Microsoft Office около двадцати).

7.3.2 Вставка текста и объектов в документ и форматирование текста

Для создания абзацев в документе можно использовать коллекцию **Paragraphs** объекта Document, представляющую собой набор абзацев данного документа.

Добавить новый абзац можно с помощью метода Add этой коллекции:

```
App.ActiveDocument.Paragraphs.Add;
```

Для вставки собственно текста в документ применяется не объект Paragraph, а объект **Range**, представляющий любую непрерывную часть документа (в том числе и вновь созданный абзац). Этот объект может быть создан разными способами. Например, можно указать начальный и конечный символы диапазона (если таковые имеются в документе):

```
var
Rng : Variant;
……….
Rng := App.ActiveDocument.Range(2,4); // со 2-го по 4-й символы
```

Можно также указать номер абзаца (например, только что созданного):

```
Rng := App.ActiveDocument.Paragraphs.Item(l).Range;
```

Кроме того, можно указать несколько последовательных абзацев:

```
Rng := App.ActiveDocument.Range(App.ActiveDocument.Paragraphs.Item(3).Range.Start,
App.ActiveDocument.Paragraphs.Item(5).Range.End)
```

Вставить текст можно с помощью методов **InsertBefore** (перед диапазоном) или **InsertAfter** (после диапазона) объекта Range, например:

```
Rng.InsertAfter('Это вставляемый текст');
```

Помимо объекта Range текст можно вставлять с помощью объекта **Selection**, являющегося свойством объекта Word.Application и представляющего собой выделенную часть документа (этот объект создается, если пользователь выделяет часть документа мышью, и может быть создан также с помощью приложения контроллера). Сам объект Selection можно создать, применив метод **Select** к объекту Range, например:

```
var
Sel : Variant;
……..
App.ActiveDocument.Paragraphs.Item(3).Range.Select;
```

В приведенном выше примере в текущем документе выделяется третий абзац.

Если мы хотим вставить строку текста в документ либо вместо выделенного фрагмента текста, либо перед ним, это можно сделать с помощью следующего фрагмента кода:

```
var
Sel: Variant;
……...
Sel := App.Selection;
Sel.TypeText('Это текст,' + ' которым мы заменим выделенный фрагмент');
```

Если свойство **Options.ReplaceSelection** объекта Word.Application равно **True**, выделенный текст будет заменен новым (этот режим действует по умолчанию); если же нужно, чтобы текст был вставлен перед выделенным фрагментом, а не вместо него, следует установить это свойство равным **False**:

```
App.Options.ReplaceSelection := False;
```

Символ конца абзаца при использовании объекта Selection может быть вставлен с помощью следующего фрагмента кода:

```
Sel.TypeParagraph;
```

К объекту Selection, так же как и к объекту Range, можно применить методы InsertBefore и InsertAfter. В этом случае, в отличие от предыдущего, вставляемый текст станет частью выделенного фрагмента текста.

С помощью объекта Selection, используя его **свойство Font** и свойства **объекта Font**, такие как Bold, Italic, Size и другие, можно отформатировать текст.

Например, таким образом можно вставить строку, выделенную полужирным шрифтом:

```
Sel.Font.Bold := True;
Sel.TypeText('Это текст, который выделен полужирным шрифтом.');
Sel.Font.Bold := False;
Sel.TypeParagraph;
```

Для наложения на вставляемый текст определенного заранее стиля можно использовать свойство **Style** объекта Selection, например:

```
Sel.Style := 'Heading 1';
Sel .TypeText('Это текст, который станет заголовком');
Sel.TypeParagraph;
```

Нередко документы Word содержат данные других приложений. Простейший способ вставить такие данные в документ — использовать метод **Paste** объекта Range:

```
var
Rng: Variant;
……
Rng := App.Selection.Range;
```

```
Rng.Collapse(wdCollapseEnd);
Rng.Paste;
```

Естественно, в этом случае в буфере обмена уже должны содержаться вставляемые данные. Если нужно поместить в буфер обмена часть документа Word, это можно сделать с помощью метода **Copy** объекта Range:

```
var
Rng: Variant;
……….
Rng := App.Selection.Range;
Rng.Copy;
```

7.3.3 Перемещение курсора по тексту

Используя метод **Collapse**, можно «сжать» объект Range или объект Selection, сократив его размер до нуля символов:

```
Rng.Collapse(wdCollapseEnd);
```

Параметр этого метода указывает, где окажется новый объект Range или Selection - в начале или в конце исходного фрагмента. Если используется позднее связывание, нужно определить в приложении соответствующие константы:

```
const
wdCollapseStart  = $00000001; // новый объект находится в начале фрагмента
wdCollapseEnd    = $00000000; // новый объект находится в конце фрагмента
```

Перемещать курсор по тексту можно с помощью метода **Move** объектов Range и Selection. Этот метод имеет два параметра. Первый из них указывает на то, в каких единицах измеряется перемещение - в символах (по умолчанию), словах, предложениях, абзацах и др. Второй параметр указывает на число единиц, на которое нужно переместиться (это число может быть и отрицательным; по умолчанию оно равно 1). Например, следующий фрагмент кода приведет к перемещению курсора на один символ вперед:

Rng.Move;

Другой фрагмент кода приведет к перемещению курсора на три абзаца вперед:

Rng.Move(wdParagraph, 3);

Отметим, что этот метод использует следующие константы:

const // Единицей перемещения является:
wdCharacter= $00000001 // символ
wdWord= $00000002 // слово
wdSentence = $00000003 // предложение
wdParagraph = $00000004 // абзац
wdStory = $00000006 //часть документа (например, колонтитул, оглавление и др.)
wdSection = $00000008 // раздел
wdColumn = $00000009 //колонка таблицы
wdRow = $0000000A //строка таблицы
wdCell = $00000000 //ячейка таблицы
wdTable= $0000000F //таблица

Нередко для перемещения но тексту используются **закладки**. Создать закладку в текущей позиции курсора можно путем добавления члена коллекции **Bookmarks** объекта Document с помощью метода Add, указав имя закладки в качестве параметра, например:

App.ActiveDocument.Bookmarks.Add('MyBookmark');

Проверить существование закладки в документе можно с помощью метода **Exists**, а переместиться на нее - помощью метода **Goto** объекта Document, Range или Selection:

Rng := App.ActiveDocument.Goto(wdGoToBookmark, wdGoToNext,,'MyBookmark');
rng. InsertAfter('Текст, вставленный после закладки');

Значения констант для этого примера таковы:
const
wdGoToBookmark = $FFFFFFFF; // перейти к закладке
wdGoToNext = $00000002: // искать следующий объект в тексте

С помощью метода **Goto** можно перемещаться не только на указанную закладку, но и на другие объекты (рисунки, грамматические ошибки и др.). Кроме этого, направление перемещения тоже может быть разным. Поэтому список констант, которые могут быть использованы в качестве параметров данного метода, довольно велик.

7.3.4 Создание таблиц

Создавать таблицы можно двумя способами.

Первый заключается в вызове метода **Add** коллекции **Tables** объекта **Document** и последовательном заполнении ячеек данными. Этот способ при позднем связывании работает довольно медленно.

Второй способ, который намного «быстрее», заключается в создании текста из нескольких строк, содержащих подстроки с разделителями (в качестве разделителя можно использовать любой или почти любой символ, но нужно, чтобы он заведомо не встречался в данных, которые в дальнейшем будут помещены в таблицу), и последующего преобразования такого текста в таблицу с помощью метода **ConvertToTable** объекта **Range**.

Ниже приведен пример создания таблицы из трех строк и трех столбцов этим способом (в качестве разделителя, являющегося первым параметром метода ConvertToTable, используется запятая):

```
var
Rng: Variant;
……….
Rng := App.Selection.Range;
Rng.Collapse(wdCol1apseEnd);
Rng.InsertAfter('l,2,3');
Rng.InsertParagraphAfter;
Rng.InsertAfter('4,5,6');
Rng.InsertParagraphAfter;
Rng.InsertAfter('7,8,9');
Rng. InsertParagraphAfter;
Rng.ConvertToTable(',');
```

Внешний вид таблицы можно изменить с помощью свойства **Format**, а также с помощью свойств коллекции **Columns**, представляющей колонки таблицы, и коллекции **Rows**, представляющей строки таблицы (объекта **Table**).

7.3.5 Обращение к свойствам документа

Свойства документа можно получить с помощью коллекции **BuiltInDocumentProperties** объекта Document, например:

```
Memol.Lines.Add('Название - ' + App.ActiveDocument,
BuiltInDocumentProperties[wdPropertyTitle].Value);
Memol.Lines.Add('Автор - ' + App.ActiveDocument,
BuiltInDocumentProperties[wdPropertyAuthor].Value);
Memol.Lines.Add('Шаблон - ' + App.ActiveDocument,
BuiltInDocumentProperties[wdPropertyTemplate].Value);
```

Константы, необходимые для обращения к свойствам документа по имени, приведены ниже:

```
const
wdPropertyTitle = $00000001; // название
wdPropertySubject = $00000002: // назначение
wdPropertyAuthor = $00000003; // автор
wdPropertyKeywords = $00000004; // ключевые слова
wdPropertyComments = $00000005; // комментарии
wdPropertyTempi ate = $00000006; // шаблон
wdPropertyLastAuthor = $00000007; // автор, последним редактировавший текст
wdPropertyRevision = $00000008; // версия
wdPropertyAppName = $00000009; // имя приложения
wdPropertyTimeLastPrinted = $0000000A; // дата последнего вывода на печать
wdPropertyTimeCreated = $0000000B; //время создания
wdPropertyTimeLastSaved = $00000000; //время последнего сохранения
wdPropertyVBATotalEdit = $00000000; //суммарное время редактирования
wdPropertyPages = $0000000E; // число страниц
wdPropertyWords = S0000000F; //число слов
wdPropertyCharacters = $00000010; // число символов
wdPropertySecurity = $00000011; // правила доступа к документу
wdPropertyCategory = $00000012; // категория
wdPropertyFormat = $00000013; // формат документа
wdPropertyManager = $00000014; // менеджер
wdPropertyCompany = $00000015; // компания
wdPropertyBytes = $00000016; //число байт
```

wdPropertyLines = $00000017; // число строк
wdPropertyParas = $00000018; // число абзацев
wdPropertySlides = $00000019; // число слайдов
wdPropertyffotes = $0000001A; // число комментариев
wdPropertyHiddenSlides = $0000001B; число скрытых слайдов
wdPropertyMMClips = $0000001C; // число мультимедиа-клипов
wdPropertyHyperlinkBase = $0000001; //путь к гипертекстовым ссылкам
wdPropertyCharsWSpaces = $0000001E; число символов без учета пробелов

7.4 Автоматизация Microsoft Excel

7.4.1 Программные идентификаторы и объектная модель Microsoft Excel

Объектная модель Microsoft Excel представлена на рисунке 7.2.

Существует три типа объектов Excel, которые могут быть созданы непосредственно с помощью приложения-контроллера. Эти объекты и соответствующие им программные идентификаторы перечислены в таблице.

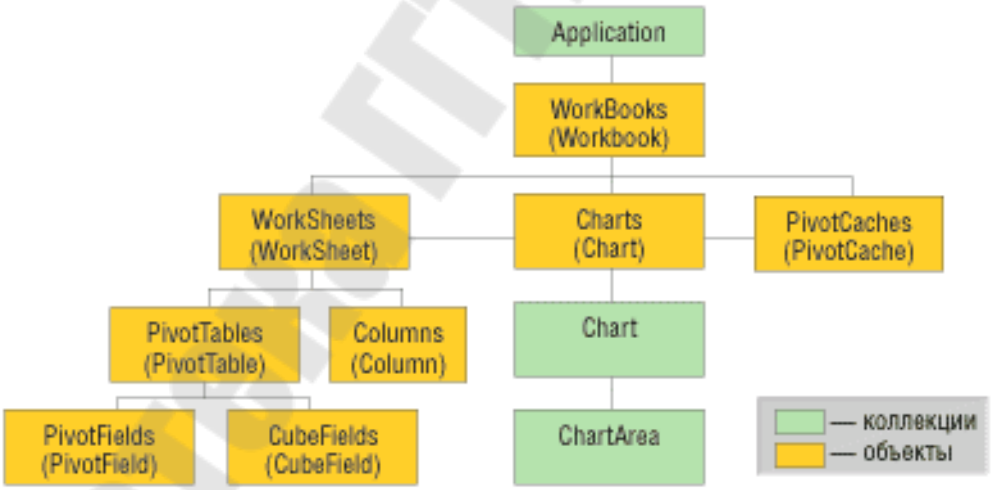

Рисунок 7.2

Таблица 7.2

Объект	Программный идентификатор	Комментарий
Application	Excel.Application	С помощью этого программного идентификатора создастся
	Excel. Application.9(10)	

		экземпляр приложения без открытых рабочих книг
WorkBook	Excel. AddIn	С помощью этого программного идентификатора создается экземпляр расширения (add-in) Excel (имеющиеся расширения доступны с помощью команды Tools ► Add-Ins)
	Excel.Chart	Рабочая книга, созданная с помощью этих программных идентификаторов, состоит из двух листов — одного для диаграммы, другого — для данных, па основе которых она построена
	Excel.Sheet	Рабочая книга, созданная с помощью этих программных идентификаторов, состоит из одного листа

Все остальные объекты Excel являются внутренними.

Основным в объектной модели Excel является объект **Application**, содержащий коллекцию **Workbooks** объектов типа **WorkBook**. Каждый объект типа **WorkBook** содержит коллекцию **Worksheets** объектов типа **Worksheet**, коллекцию **Charts** объектов типа **Chart** и др.

Манипуляция рабочими книгами, их листами, ячейками, диаграммами реально осуществляется путем обращения к свойствам и методам этих объектов.

Рассмотрим наиболее часто встречающиеся задачи, связанные с автоматизацией Microsoft Excel. Более полная информация приводится в справочном файле VBAXL9.CHM (или VBAXL10.CHM).

7.4.2 Запуск Microsoft Excel, создание и открытие рабочих книг

При разработке примеров использования Microsoft Excel можно задействовать код создания контроллера, приведенный в начале лекции, заменив первую строку кода в приведенном там примере следующей:

```
AppProgID := 'Excel.Application';
```

Кроме того, нужно заменить комментарии кодом, манипулирующим свойствами и методами объекта Excel.Application. Однако, подключение контроллера автоматизации к имеющейся версии Excel с помощью метода GetActiveOleObject может привести к тому, что вся клиентская часть Excel окажется невидимой (это происходит, если имеющаяся копия Excel запущена в режиме, когда ее пользовательский интерфейс недоступен). Поэтому лучше упростить код создания контроллера и всегда создавать новую копию Excel.

Создать новую рабочую книгу Excel можно, используя метод **Add** коллекции Workbooks объекта Application:

App.WorkBooks.Add;

Для создания рабочей книги на основе шаблона следует указать его имя в качестве первого параметра метода Add:

App.WorkBooks.Add('C:\Program Fi les\Microsoft Office\Templates\1033\invoice.xlt');

В качестве первого параметра этого метода можно также использовать следующие константы:

```
const
xlWBATChart      = IFFFFEFF3; // рабочая книга состоит из листа с диаграммой
xlWBATWorksheet = IFFFFEFB9; // рабочая книга состоит из листа с данными
```

В этом случае рабочая книга будет содержать один лист типа, заданного указанной константой (график, обычный лист с данными и др.).

Для открытия существующего документа следует воспользоваться методом **Open** коллекции WorkBooks:

App.Documents.Open('C:\MyExcelFile.xls');

Отметим, что свойство ActiveWorkBook объекта Excel.Application указывает на текущую активную рабочую книгу среди одной или нескольких открытых. Помимо этого, к рабочей книге можно обращаться по ее порядковому номеру. Например, ко второй открытой рабочей книге можно обратиться так:

App.WorkBooks[2]

Следует обратить внимание на то, что при позднем связывании синтаксис обращения к членам коллекций объектов Excel отличен от синтаксиса обращения к объектам Word. В случае Word мы использовали метод Item, а в случае Excel мы обращаемся к членам коллекции как к элементам массива.

Сделать рабочую книгу активной можно с помощью метода **Activate**:

App.WorkBooks[2].Activate:

7.4.3 Сохранение, печать и закрытие рабочих книг Microsoft Excel

Закрытие документа может быть осуществлено с помощью метода Close одним из следующих способов:

App.WorkBooks[2].Close;
App.ActiveWorkBook.Close;

В случае позднего связывания метод **Close** имеет несколько необязательных параметров, влияющих на правила сохранения рабочей книги. Первый из параметров принимает значения True или False и определяет необходимость сохранения изменений, внесенных в рабочую книгу. Второй параметр (типа Variant) — это имя файла, в котором нужно сохранить рабочую книгу (если в нее были внесены изменения). Третий параметр, также принимающий значения True или False, определяет необходимость пересылки документа следующему пользователю по электронной почте и может быть проигнорирован, если эта функциональность не требуется.

App.ActiveWorkBook.Close(True,'C:\MyWorkBook.xls');

Просто сохранить рабочую книгу, не закрывая ее, можно с помощью метода **Save**:

App.ActiveWorkBook.Save;

Используется также метод **SaveAs**:

App.ActiveWorkBook.SaveAs('C:\MyWorkBook.xl s');

Метод SaveAs имеет более десятка параметров, влияющих на то, как именно сохраняется документ (под каким именем, с паролем или без него, какова кодовая страница для содержащегося в ней текста и др.).

Закрыть само приложение Excel можно с помощью метода **Quit** объекта Excel.Application. В случае Excel этот метод параметров не имеет.

Вывод документа Excel на устройство печати можно осуществить с помощью метода **Printout** объекта WorkBook, например:

App.ActiveWorkBook.PrintOut;

Если нужно изменить параметры печати, следует указать значения соответствующих параметров метода PrintOut (в случае Excel их восемь).

7.4.4 Обращение к листам и ячейкам

Обращение к листам рабочей книги производится с помощью коллекции **WorkSheets** объекта WorkBook. Каждый член этой коллекции представляет собой объект **Worksheet**. К члену этой коллекции можно обратиться по его порядковому номеру, например:

App.WorkBooks[l].WorkSheets[l].Name:= Страница Г';

Приведенный выше пример иллюстрирует, как можно изменить имя листа рабочей книги. К листу рабочей книги можно обратиться и по имени, например:

App.WorkBooks[l].WorkSheets['Sheetl'].Name := 'Страница 1';

Обращение к отдельным ячейкам листа производится с помощью коллекции **Cells** объекта Worksheet. Например, добавить данные в ячейку B1 можно следующим образом:

App.WorkBooks[l].WorkSheets['Sheetl'].Cells[l,2].Value := '25';

Примечание: Здесь первая из координат ячейки указывает на номер строки, вторая — на номер столбца.

Добавление формул в ячейки производится аналогичным способом:

```
App.WorkBooks[l].WorkSheets["Sheetl" ].Cells[3.2].Value
:='=SUM(B1:B2)';
```

Очистить ячейку можно с помощью метода **ClearContents**.

Форматирование текста в ячейках производится с помощью свойств **Font** и **Interior** объекта Cell и их вложенных свойств. Например, следующий фрагмент кода выводит текст в ячейке красным полужирным шрифтом Courier кегля 16 на желтом фоне:

```
App.WorkBooks[l].WorkSheets[l].Cells[3,2].Interior.Color := clYellow:
App.WorkBooks[l].WorkSheets[l].Cells[3,2].Font.Color := clRed;
App.WorkBooks[l].WorkSheets[l].Cells[3.2].Font.Name := 'Courier1';
App.WorkBooks[l].WorkSheets[l].Cells[3,2].Font.Size := 16;
App.WorkBooks[l].WorkSheets[l].Cells[3.2].Font.Bold := True;
```

Вместо свойства **Color** можно использовать свойство ColorIndex, принимающее значения от 1 до 56: таблицу соответствий значений этого свойства реальным цветам можно найти в справочном файле VBAXL9.CHM или VBAXL10.CHM.

Обратиться к текущей ячейке можно с помощью свойства **ActiveCel1** объекта Excel.Application, а узнать местоположение ячейки — с помощью свойства **Address** объекта Cel1, например:

```
ShowMessage(App.ActiveCell.Address);
```

Помимо обращения к отдельным ячейкам, можно манипулировать прямоугольными областями ячеек с помощью объекта **Range**, например:

```
App.WorkBooks[l].WorkSheets[2].Range['Al:C5'].Value := 'Test';
App.WorkBooks[l].WorkSheets[2].Ranger['A1:C5'].Font.Color := clRed;
```

Приведенный выше код приводит к заполнению прямоугольного участка текстом и изменению цвета шрифта ячеек.

Объект **Range** также часто используется для копирования прямоугольных областей через буфер обмена. Ниже приведен пример, иллюстрирующий такую возможность:

```
App.WorkBooks[l].WorkSheets[2].Range['A1:C5'].Copy;
App.WorkBooks[l].WorkSheets[2].Range['All:C15'].Select;
App.WorkBooks[l].WorkSheets[2].Paste;
```

Обратите внимание на то, что диапазон, куда копируются данные, предварительно выделяется с помощью метода **Select**.

7.4.5 Создание диаграмм

Диаграммам Excel соответствует объект **Chart**, который может располагаться как на отдельном листе, так и па листе с данными. Если объект Chart располагается на листе с данными, ему соответствует член коллекции **ChartObjects** объекта **Worksheet**, и создание диаграммы нужно начать с добавления элемента в эту коллекцию:

```
Ch := App.WorkBooks[l]. WorkSheets[2]. ChartObjects. Add(0,50,400,400);
```

Параметрами этого метода являются координаты левого верхнего угла и размеры диаграммы в пунктах (1/72 дюйма).

Если же диаграмма располагается на отдельном листе (не предназначенном для храпения данных), то ее создание нужно начать с добавления элемента в коллекцию **Sheets** объекта Application (отличающуюся от коллекции Worksheets тем, что она содержит листы всех типов, а не только листы с данными):

```
App.WorkBooks[l].Sheets.Add( , ,1 ,xlWBATChart);
```

В этом случае первый параметр метода Add идентифицирует порядковый номер листа, перед которым нужно поместить лист с диаграммой (или листы, если их несколько), второй параметр — порядковый номер листа, после которого нужно поместить лист с диаграммой (используется обычно один из них), третий параметр — количество создаваемых листов, а четвертый — их тип. Значения четвертого параметра совпадают со значениями первого параметра метода Add коллекции WorkBooks объекта Application, и при использовании имен соответствующих констант следует определить их в приложении-контроллере.

Простейший способ создать диаграмму с точки зрения пользователя — построить ее с помощью соответствующего мастера на основе прямоугольной области с данными. Точно так же можно создать диаграмму и с помощью контроллера автоматизации — для этой цели у объекта Chart, являющегося свойством объекта ChartObject (члена коллекции ChartObjects), имеется метод **ChartWizard**.

Первым параметром этого метода является объект Range, содержащий диапазон ячеек для построения диаграммы, а вторым —

числовой параметр, указывающий, какого типа должна быть эта диаграмма:

```
var
Ch: Variant;
……
Ch.Chart.ChartWizard(App.WorkBooks[l].WorkSheets[2].Range['Al:C5']
,xl3DColumn);
```

Возможные значения параметра, отвечающего за тип диаграммы, можно найти в справочном файле.

У объекта Chart имеется множество свойств, которые отвечают за внешний вид диаграммы и с помощью которых можно изменить ее точно так же, как это делают пользователи вручную. Ниже приводится пример создания заголовка диаграммы и подписей вдоль ее осей (оси есть не у всех типов диаграмм):

```
Ch.Chart.HasTitie              := 1;
Ch.Chart.HasLegend             := False;
Ch.Chart.ChartTitle.Text       := 'Пример диаграммы Excel';
Ch.Chart.Axes(l).HasTitle      := True;
Ch.Cha rt.AxesA).Axi sTi tle.Text  := 'Подпись вдоль оси абсцисс'
Ch.Chart.AxesB).HasTitle       := True;
Ch.Chart.AxesB).AxisTitie.Text := 'Подпись вдоль оси ординат'
```

Еще один способ создания диаграммы — определить все ее параметры с помощью свойств объекта Chart, включая и определение серий, на основе которых она должна быть построена. Данные для серии обычно содержатся в объекте Range, содержащем строку или столбец данных, а добавление серии к диаграмме производится путем добавления члена к коллекции SeriesCollection, например:

```
App.WorkBooks[l].Sheets.Add( , , 1, xlWBATChart);
App.WorkBooks[l].Sheets[l].ChartType := xl3DPie;
Rng:=App.WorkBooks[1].Worksheets[2].Range['Bl:B5'];
App.WorkBooks[l].Sheets[l].SeriesCollection.Add(Rng);
```

В данном примере к диаграмме, созданной на отдельном листе, специально предназначенном для диаграмм, добавляется одна серия на основе диапазона ячеек другого листа.

7.4.6 Применение средств доступа к данным

Из коллекций объекта Worksheet, позволяющих обращаться к реляционным и OLAP- данным с помощью средств **Microsoft Query**, наиболее интересны коллекции **QueryTables** и **PivotTables**, а также коллекция **PivotCaches** объекта Workbook. С помощью этих коллекций можно отображать в Excel результаты запросов к базам данных, доступных с помощью механизмов доступа к данным ODBC и OLEDB.

Отобразить на рабочем листе Excel результат запроса к базе данных можно, добавив новый объект QueryTable к коллекции QueryTables объекта Workbook и установив его свойство **CommandText** равным тексту запроса, например:

```
var
App, QTable: Variant;
……….
            QTable:= App.WorkBooks[1].Sheets[1].QueryTables.Add
      ('ODBC; DRIVER=SQL Server;'+
      'SERVER=MAINDESK;UID=Administrator; + 'APP=Microsoft
      Office
      XP;WSID=MAINDESK;'+'DATABASE=Northwind:Trusted_C
      onnecti on=Yes', Ws.Range['Al:Al']);
QTable.CommandText := 'SELECT * FROM Customers';
QTable.Refresh;
```

При необходимости можно отобразить имена полей, установив соответствующее значение свойства FieldNames объекта QueryTable:

```
QTable.FieldNames := True;
```

Можно также привести ширину колонок в соответствие с длиной полей набора данных, полученного в результате запроса:

```
QTable.AdjustColumnWidth := True;
```

Для создания сводных таблиц следует осуществить кэширование просуммированных данных в памяти Excel. Для этого перед созданием сводной таблицы нужно создать объект **QueryTablePivotCache**, добавить его к коллекции PivotCaches объекта Workbook и установить его свойства **Connection**, **CommandType** и **CommandText**, определяющие, откуда берутся исходные данные для сводной таблицы:

```
var
WB, PC, PT: Variant;
……..
PC := WB.PivotCaches.Add(xlExternal);
PC.Connection := 'OLEDB;Provider=Microsoft.Jet.0LEDB.4.0;' +
'Data Source=C:\data\Northwind.mdb';
PC.CommandType := xlCmdSql;
PC.CommandText := 'SELECT Country. City. '+
' ProductName. Salesperson. ExtendedPrice FROM Invoices ';
```

Далее следует создать саму сводную таблицу:

```
PC.CreatePivotTable(WB.Worksheets[l].Cells[l,l],'PivotTablel');
PT := WB.Worksheets[l].PivotTables('PivotTablel');
```

В этом случае нам потребуются следующие константы:

```
const
xlExternal = $00000001;//используются внешние данные
xlCmdSql = $00000002; //используется SQL-запрос
```

Можно предоставить конечному пользователю возможность самому определить, как будут расположены поля сводной таблицы, а можно расположить их программно, воспользовавшись коллекцией **PivotFields** объекта PivotTable:

```
PT.PivotFields('Country).Orientation := xlRowField;
PT.PivotFields('ProductName').Orientation := xlPageField;
PT.PivotFields('Salesperson).Orientation := xlColumnField;
PT.PivotFields('ExtendedPrice).Orientation := xlDataField;
```

В этом случае нам потребуются следующие константы:

```
const
xl Hidden = $00000000: // поле не используется
xlRowField = $00000001; // поле помещается в область строк
xlColumnField = $00000002; // поле помещается в область столбцов
xlPageField = $00000003; // поле помещается в область фильтров
xlDataField = $00000004; // поле помещается в область суммируемых данных
```

Список использованных источников

1. Delphi и технология COM. Мастер-класс / Н. Елманова, С. Трепалин, А. Тенцер. – СПб.: Питер, 2003. – 698 с.
2. Архангельский А.Я., ПрограммированиевDelphi7. – М.: ООО «Бином-Пресс», 2003. – 1152с.
3. Дейл Роджерсон, Основы COM. – М. Русская редакция, 2000 – 400с.
4. КорняковВ.Н., Программирование документов и приложений MSOffice в Delphi. – СПб.:БХВ-Петербург,2005. – 496с.
5. Сухарев М.В., Основы Delphi. Профессиональный подход – СПб.: Наука и Техника, 2004. – 600с.

Содержание

Самовендюк Николай Владимирович

ТЕХНОЛОГИИ КОМПОНЕНТНОГО ПРОГРАММИРОВАНИЯ

ПОСОБИЕ
по одноименному курсу
для слушателей специальности 1-40 01 73
«Программное обеспечение информационных систем»
заочной формы обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 28.11.16.
Рег. № 92Е.
http://www.gstu.by