Федеральное агентство по образованию
Озёрский технологический институт —
филиал НИЯУ МИФИ

Вл. Пономарев

Введение в ActiveX

Учебно-методическое пособие
по дисциплине
«Современные технологии программирования»

Озёрск — 2012

УДК 681.3.06
П56

Пономарев В.В. Введение в ActiveX. Учебно-методическое пособие по дисциплине «Современные технологии программирования». Редакция 08.08.2012. Озёрск: ОТИ НИЯУ МИФИ, 2012. — 120 с., ил.

Пособие предназначено для изучения инфраструктуры повторно используемых объектов Microsoft COM студентами, обучающимися по специальности «Программное обеспечение вычислительной техники и автоматизированных систем».

Рецензенты:
1) Начальник отдела ПО ЗАО «Озёрск Телеком» Н.Г. Ведюшкин.
2) Ст. преподаватель кафедры ПМ ОТИ НИЯУ МИФИ Р.А. Федотов.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Обозначения и сокращения

ATL - ActiveX Template Library.
COM - Component Object Model.
MSVB - Microsoft Visual Basic 6.0.
MSVS - Microsoft Visual Studio 2008.

Введение

Цель настоящего пособия — дать основные сведения о технологиях Microsoft, основанных на COM. Развитие этих технологий в конечном итоге привело к созданию самой совершенной на сегодняшний день технологии от Microsoft — .NET. Поэтому изучение технологий ActiveX представляется логичным продолжением изучения концепций объектно-ориентированного программирования.

Пособие состоит из пяти основных глав — «Основы COM», «Инфраструктура COM», «Использование ATL 3.0», «COM+» и «Автоматизация (OLE Automation)».

В главе «Основы COM» рассматриваются базовые понятия о COM-объектах, интерфейсах, технологиях, основанных на COM.

В главе «Инфраструктура COM» рассматриваются все основные элементы инфраструктуры COM, необходимые для понимания принципов функционирования COM-объектов — стандартные интерфейсы, типы серверов, потоковая модель, маршалинг, библиотека типов и т.п.

В главе «Использование ATL 3.0» кратко описывается процесс создания сервера COM средствами ATL Microsoft Visual Studio.

В главе «COM+» приведены основные сведения о технологии COM+ для получения общего представления. Там же описывается практическая работа, позволяющая увидеть COM+ в действии.

В главе «Автоматизация (OLE Automation)» рассматривается программное управление сервисами, предоставляемыми установленным на компьютере программным обеспечением. Эта технология поддерживается и является актуальной до сих пор.

В главе «Дополнительные материалы» рассматриваются строковые типы системы Windows и базовые типы автоматизации.

В главе «Рекомендуемая литература» приводятся литературные источники, рекомендуемые для более детального изучения технологии повторно используемых COM-объектов.

Основы COM

Одной из самых важных и сложных задач в программировании является повторное использование кода (*code reuse*). На протяжении многих десятилетий программы в большинстве случаев создаются «с нуля», в то время как в других отраслях промышленного производства используются те или иные «строительные компоненты». Производители компьютеров используют, например, микросхемы, транзисторы, конденсаторы и т.п., строители — кирпичи, блоки, панели, балки и т.д., машиностроители — болты, гайки, шарикоподшипники…

В программировании же практически не существует готовых к использованию компонентов, с помощью которых можно было бы «складывать» программы так, как каменщик выкладывает арку из кирпичей. Если бы компьютеры производились так, как пишутся программы, то сначала нужно было бы добыть песок, из которого можно получить кремниевую пластину, на которой затем нужно было бы сформировать необходимые p-n переходы, составляющие основу логических схем и т.д., … долгий процесс.

Нельзя сказать, что в программировании все уж так плохо. Существует много технологий, методологий и средств программирования, которые в значительной степени помогают быстро создавать качественное программное обеспечение. Здесь можно отметить объектно-ориентированное программирование, шаблоны, CASE средства на основе UML, элементы управления VB (vbx) и многое другое.

Тем не менее проблема заключается в том, что большинство технологий в конечном итоге сводятся к повторному использованию текстов программ, а не бинарных (двоичных) компонентов. Например, с появлением методологии объектно-ориентированного программирования возникли надежды на широкое повторное использование классов, однако на самом деле оказалось, что существует множество технических и иных нюансов, которые не позволили этой технологии изменить подход к конструированию программ в той мере, в какой изобретение микросхем изменило подход к конструированию электронной аппаратуры.

Известно, например, что при повторном использовании текстов программ срабатывает так называемый эффект NIH (not invented here, изобретено не здесь). В результате готовые тексты подвергаются модернизации, приспосабливанию к условиям конкретного проекта.

COM-объекты появились в процессе развития операционной системы Windows. Отличительными особенностями Windows является многозадачность и ориентированность на документ. Многозадачность позволяет пользователю работать сразу с несколькими документами одновременно. Ориентированность на документ (document-centric operating

system) отражает идеологическую направленность Windows на документ (данные приложения), как на главное, основное понятие в работе пользователя.

До появления Windows операционные системы, типа MS-DOS, были ориентированными на приложение. Это означает, что для работы с информацией определенного типа пользователь должен сначала открыть (запустить) приложение, а потом с его помощью открыть (загрузить) документ для работы с ним.

В Windows, напротив, пользователь ищет документ и «открывает» его. Операционная система сама выбирает приложение, которое может обработать информацию, содержащуюся в документе. Такой способ работы с информацией более точно отражает практический опыт пользователя.

Стремясь обеспечить максимум удобств для работы с документами, разработчики Windows внедрили в нее средства для соединения информации разного рода в одном документе. Первоначальная версия этих средств под названием OLE (читается olay) оказалась громоздкой, неуклюжей и требующей значительные ресурсы компьютера.

OLE расшифровывается как «связь и внедрение объектов» (*object linking and embedding*). На самом деле OLE реализуется в виде функций копирования, вырезания и вставки. Выделив и скопировав информацию одного вида в разработанный для этой цели буфер обмена (*clipboard*), ее можно затем вставить в документ, изначально содержащий информацию другого вида. То, что вставляется в документ, стали называть объектом. В зависимости от того, каким образом инородная информация привязывается к документу, различают внедрение или связь объекта. Если объект находится в файле документа, он называется внедренным в него, а если объект находится в отдельном файле — то связанным (документ содержит ссылку на этот файл).

Поскольку инородная информация не может быть обработана приложением документа, она и не обрабатывается ею. Вместо этого объект обрабатывается тем приложением, которым он был создан (или другим подходящим приложением, «прописанным» в реестре Windows). Практически пользователь, например, дважды щелкает на объект, и происходит запуск приложения, которое предназначено для обработки информации объекта.

Стремясь улучшить технологию OLE, разработчики Microsoft PowerPoint (1993) поставили перед собой важнейший, как оказалось впоследствии, вопрос — каким образом один программный компонент предоставляет свои услуги (*сервисы*) другому программному компоненту? Ответом на этот вопрос стало рождение COM-объектов, которые пе

ревернули представление о том, как должны взаимодействовать между собой программные компоненты.

Старую технологию OLE стали называть OLE1, а новую — OLE2 или просто OLE. Появившиеся на основе COM-объектов технологии для Интернет фирма Microsoft объединила под знаменем ActiveX. Хотя изначально технологии ActiveX предназначены для Интернет, теперь этим термином пользуются для обозначения взаимоотношений программных компонентов на основе COM-объектов.

Понятие о COM

Каким образом одна часть ПО получает доступ к сервисам, предоставляемой другой частью? Ответ зависит от того, что представляют собой эти части. Модель взаимодействия ПО до использования COM представлена на рисунке 1.

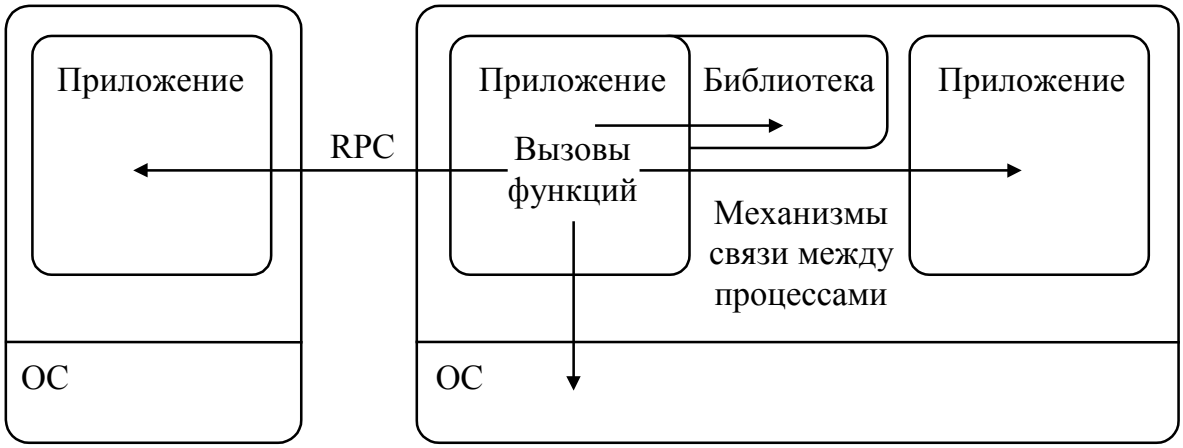


Рисунок 1 - Взаимодействие программных компонентов до COM

Приложение, скомпонованное с библиотекой, использует ее сервисы посредством вызова функций. Сервисы ОС предоставляются через системные функции. Сервисы программ, выполняющихся на этой машине, можно через механизмы связи между процессами. Получить доступ к сервисам ПО, выполняющегося на другой машине, можно посредством обмена сообщениями с удаленным приложением или вызовом удаленной процедуры (RPC — *Remote Procedure Call*). В принципе, проблема одна: как одна часть ПО получает сервисы другой, но методы разные — системный вызов, вызов функции, вызов удаленной процедуры и т.п.

COM определяет стандартный механизм предоставления сервисов. Общая архитектура сервисов в библиотеках, приложениях, системном и сетевом ПО позволяют COM изменить подход к созданию программ.

Как работает COM

В COM любая часть ПО реализует свои сервисы как один или несколько объектов COM (объекты отличаются от тех, что в Си++). Каждый такой объект поддерживает один или несколько интерфейсов, состоящих из методов. Метод — это функция (процедура), выполняющая некоторое действие, которая может быть вызвана программным обеспечением, использующим данный объект, — клиентом объекта.

Методы, составляющие каждый отдельный интерфейс, обычно являются взаимосвязанными. Клиенты могут получить доступ к сервисам объекта COM только через вызовы методов интерфейса объекта. Доступа к данным объекта у них нет. Клиент, таким образом, полностью абстрагирован от элементов данных объекта и от его устройства. Объект, в свою очередь, полностью инкапсулирован и представляет собой черный ящик.

Представим себе корректор орфографии Speller, реализованный в виде объекта COM. Такой объект может поддерживать интерфейс *ISpellCheck*, включающий методы *LookupWord*, *AddToDictionary*, *RemoveFromDictionary*. Если позднее разработчик захочет добавить к этому объекту поддержку словаря синонимов, то объекту потребуется еще один интерфейс IThesaurus, возможно, с единственным методом *GetSynonim*. Методы каждого из интерфейсов сообща предоставляют связанные друг с другом сервисы.

Большинство объектов COM поддерживают несколько интерфейсов, из которых как минимум один является стандартным служебным интерфейсом объекта, предоставляющим базовые, служебные, сервисы любого объекта, а другие интерфейсы предоставляют функциональность объекта, его основные сервисы (custom-интерфейсы). Стандартный интерфейс принято обозначать точкой (ручкой), направленной вверх. Сам объект всегда располагается внутри некоторого сервера (рисунок 2), который может быть либо динамически подключаемой библиотекой DLL, либо отдельным, самостоятельным процессом (приложением).

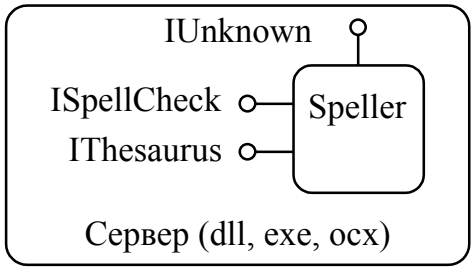


Рисунок 2 - Корректор орфографии — Объект COM

Если поближе рассмотреть, к примеру, интерфейс *ISpellCheck*, то мы увидим его методы *LookupWord*, *AddToDictionary* и *RemoveFromDictionary*, а также несколько дополнительных методов, добавляемых автоматически (рисунок 3, дополнительные методы не показаны).

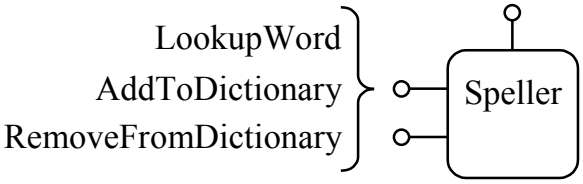


Рисунок 3 - Методы интерфейса ISpellCheck

Чтобы обратиться к методам интерфейса объекта COM, клиент должен получить указатель на этот интерфейс, для каждого интерфейса свой указатель (рисунок 4, указатели — черные точки).

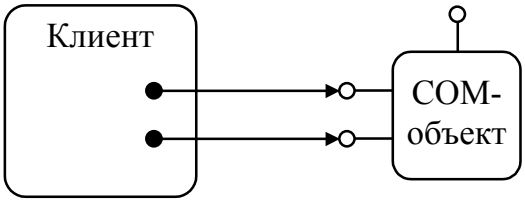


Рисунок 4 - Клиент использует указатели для доступа к интерфейсу

Любой объект COM — это экземпляр определенного класса. Знать класс объекта необходимо для запуска экземпляра объекта, выполняемого при помощи библиотеки COM. Эта библиотека присутствует на любой системе, поддерживающей COM, и имеет доступ к справочнику всех доступных на данной машине классов COM-объектов. Клиент может вызвать функцию библиотеки COM, передав ей класс нужного ему COM-объекта и задав один из поддерживаемых объектом интерфейсов, указатель на который нужен клиенту в первую очередь. (Вызовы библиотеки COM реализованы в виде обычных функций, а не в виде интерфейсов COM, чтобы не возникло замкнутого круга.) Затем библиотека запускает сервер, реализующий объект данного класса и возвращает клиенту указатель запрошенного интерфейса нового экземпляра класса. Далее клиент может запросить указатель на другой интерфейс непосредственно у самого объекта.

Получив указатель на нужный ему интерфейс выполняющегося объекта, клиент использует сервисы объекта, вызывая методы интерфейса через полученные указатели. С точки зрения программиста, вызов метода аналогичен вызову локальной функции. На самом деле код, выполняющийся по вызову метода, может быть частью библиотеки либо отдельного процесса и даже вообще располагаться на другой машине.

Доступ ко всем сервисам осуществляется единообразно. Для доступа к сервисам, предоставляемыми любыми типами ПО, используется одна общая модель (рисунок 5).

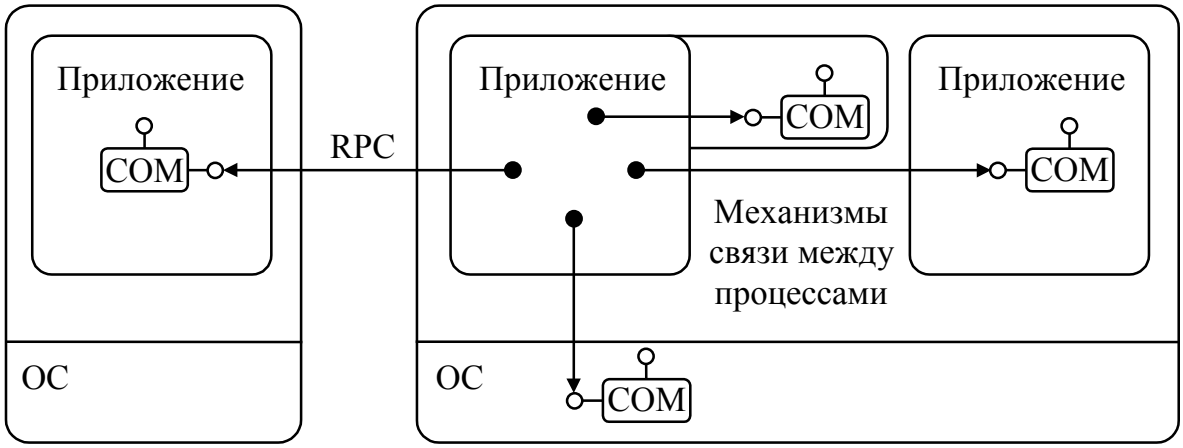


Рисунок 5 - Модель сервисов COM

COM и объектно-ориентированный подход

Чтобы считать технологию объектно-ориентированной, она должна поддерживать инкапсуляцию, наследование и полиморфизм.

В классическом ООП объект представляет собой набор данных и методов для управления ими. Объекты COM являются таковыми. Главное отличие заключается в том, что доступ к данным объекта COM осуществляется через методы, которые в этом случае называются свойствами. Для доступа к одному свойству объекта COM требуется два метода — один для установки нового значения свойства, другой — для чтения. Объект COM представляется нам полностью инкапсулированным.

В большинстве ООП технологий объект поддерживает только один интерфейс. Так, интерфейсом объекта Си++ является все, что объявлено в открытой секции описания класса. Объект COM, напротив, почти всегда поддерживает несколько интерфейсов. COM-объект, поддерживающий только один интерфейс, практически бесполезен, потому что в этом случае интерфейс является стандартным IUnknown, который не предназначен для предоставления какой-либо функциональности, кроме служебной. Кроме того, один COM-объект может быть реализован, например, несколькими объектами Си++.

Класс в COM понимается как конкретная реализация набора интерфейсов. Может существовать несколько различных реализаций одного и того же набора интерфейсов, каждая из которых будет отдельным классом. С точки зрения клиента только интерфейс имеет значение. Возможность работать с объектами разных типов, каждый из которых под

держивает данный набор интерфейсов, но реализует их по-разному, называется полиморфизмом. Полиморфизм означает, что клиент может рассматривать разные объекты как одинаковые. Эту идею COM реализует в полной мере. Более того, все объекты COM являются полиморфными по отношении друг к другу — они все поддерживают стандартный интерфейс IUnknown.

Класс Си++ может быть производным от другого класса. Классическое ООП поддерживает наследование реализации — повторное использование кода. Существует также наследование интерфейса — повторное использование спецификации. В Си++ наследование интерфейса осуществляется при помощи абстрактных классов. Технология COM предполагает именно наследование интерфейса. Полиморфизм в COM обеспечивается наследованием того или иного интерфейса, и реализацией его в виде объекта того или иного класса. Объект COM всегда содержит новый код.

Классы Си++ могут образовывать иерархию. В COM иерархия отсутствует. Вместо иерархии классов в COM используется включение одних объектов в другие. В Си++ класс может не использовать наследование, в COM объект всегда наследует множество интерфейсов — это основополагающий принцип. В COM новый интерфейс может наследовать старый, и объект COM, поддерживающий новый интерфейс, одновременно гарантирует поддержку старого.

В классическом ООП класс есть описание типа, существующее только на момент трансляции. В COM описание интерфейса является спецификацией, существенно необходимой в разные моменты существования COM-объекта: проектирование, компиляция, исполнение. Интерфейсы хранятся в отдельных файлах, в так называемых библиотеках типов, имеющих расширение .tlb и .olb.

COM и многокомпонентные программы

Конструкторам программ не хватает компонентов. Создание новых приложений на основе существующих протестированных компонентов должно приводить к более надежному коду быстрее и дешевле.

Объекты COM являются эффективным механизмом повторного применения ПО, т.к. они позволяют создавать дискретные повторно используемые бинарные компоненты.

Идея не нова. Рассмотрим библиотеки. Они привычны и просты в использовании. Недостатки — сложно расширить функциональные возможности. Как установить новую версию, не повредив старым приложениям? Как установить в системе несколько реализаций?

Другой пример — объекты. Они дают больше возможностей, чем библиотеки, благодаря полиморфизму. Однако почему нет рынка классов?

1. Отсутствует стандарт для компоновки двоичных объектов;
2. Объекты Си++ нельзя использовать в программе на ObjectPascal;
3. Изменения в одном объекте потребует перекомпоновки, а то и перекомпиляции проекта. Если объект используется несколькими приложениями, потребуется все их перекомпилировать.

COM решает все проблемы. Он предоставляет стандарт на двоичный формат, в котором распространяются объекты COM. Т.к. экземпляры объектов COM создаются тогда, когда это необходимо, то после установки в системе новой версии объекта все его клиенты автоматически получают новый вариант объекта. Главный плюс — универсальный метод доступа ко всем типам программных сервисов.

Основные преимущества COM

1. Преимущества ООП;
2. COM сглаживает различие между системным и прикладным ПО;
3. COM не зависит от языка программирования;
4. Простой подход к контролю версий.

Обзор технологий COM

Автоматизация (OLE Automation)

Электронные таблицы, текстовые процессоры и другие программы предоставляют все виды полезных сервисов. Почему бы не обеспечить доступ к ним и другому программному обеспечению? Чтобы это стало возможным, приложения должны предоставлять свои сервисы не только пользователю, но и программам — они должны быть программируемыми. Обеспечение программируемости является целью автоматизации (первоначально называвшейся OLE-автоматизацией).

Приложение можно сделать программируемым, обеспечив доступ к его сервисам через обычный COM-интерфейс. Однако так поступают редко. Вместо этого доступ к сервисам приложений осуществляется через диспинтерфейс (*dispinterface*) IDispatch. Он отличается унифицированным способом вызова методов *custom*-интерфейса, и может быть использован в простых языках вроде *Visual Basic* и в скриптах.

Перманентность (Persistence)

Объекты состоят из методов и данных. Многим объектам необходимо сохранять свои данные в течение периодов неактивности, т.е. сде

лать их перманентными (*persistent*). Объекты COM используют для этого структурированное хранилище (*Structured Storage*). Структурированное хранилище — это файловая система внутри файла, предназначенная для сохранения данных разных приложений. Структурированное хранилище определяет два типа COM-объектов: *Storage* (хранилище, интерфейс IStorage) и *Stream* (поток, интерфейс IStream). На рисунке 6 приведен примерный вид структурированного хранилища.

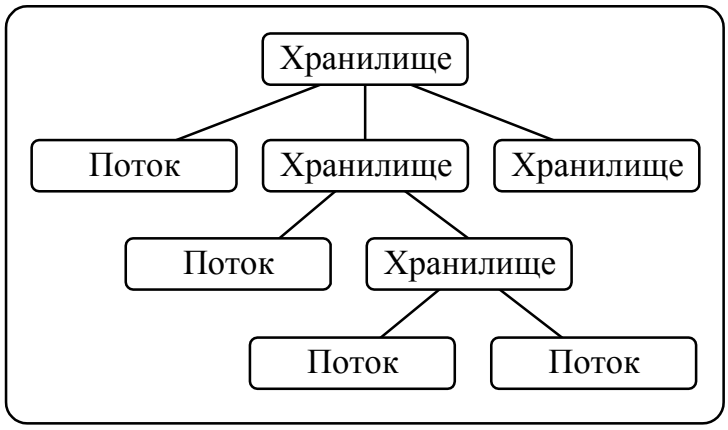


Рисунок 6 - Структурированное хранилище

Используемые для обеспечения перманентности данных интерфейсы имеют вид IPersistXXXX, например, IPersistStorage, IPersistFile, IPersistPropertyBag и др.

Моникеры (moniker)

Моникер — это COM-объект специфического назначения. Любой моникер «знает», как создать и инициализировать экземпляр конкретного COM-объекта. Он скрывает детали инициализации COM-объекта от клиента, «облегчая» его жизнь. Использует интерфейс IMoniker. Моникеры могут создаваться для файлов, внедренных в документ объектов и для других целей.

Единообразная передача данных (uniform data transfer)

Обмен данными — фундаментальная операция в программировании. Методы ее интерфейсов определяют стандартные способы для описания передаваемых данных (указания их местоположения, формата), собственно пересылки и обратной связи для уведомления об изменении. Основным интерфейсом является IDataObject. Единообразная передача данных используется для работы с системным буфером обмена (clipboard), в технологии *Drag-And-Drop*, в составных документах.

Технология объектов с подключениями

Технология объектов с подключением (connectable objects) обеспечивает общий механизм обратной связи объекта с клиентом, позволяя клиенту получать уведомления о событиях. Со стороны клиента используется интерфейс IConnectionPoint, со стороны сервера — IConnectionPointContainer. В *Visual Basic 6.0* объектам с подключениями соответствуют события. Эта технология предполагает «подписку» на исходящие от объекта вызовы. Объект, обращаясь к исходящему интерфейсу, вызывает функции интерфейса во всех «подписавшихся» объектах.

Составные документы

При создании составного документа одно из приложений всегда является контейнером. Другие приложения (серверы) могут размещать свои документы внутри документа-контейнера. Связанный документ хранится в отдельном файле, а в документе-контейнере хранится лишь моникер. Внедренный документ хранится в том же файле, что и документ-контейнер, а приложения совместно используют общий файл при помощи структурированного хранилища. Для обеспечения функциональности составных документов используется большое количество интерфейсов, в основном имеющих вид IOleXXXX, например, IOleObject, IOleClientSite, IOleInPlaceObject, IViewObject и др.

Управляющие элементы ActiveX

Известны под названиями элементы VBX, OLE или OCX. Это серверы-контейнеры элементов управления, многократно используемых для создания интерфейса приложения. Используют интерфейс IDispatch. Особенностью является наличие так называемых «страниц свойств» (property page), которые могут быть использованы для управления свойствами элементов во время разработки проекта. Элемент управления обязан наследовать только интерфейс IUnknown, однако на практике элементы наследуют больше интерфейсов, чем COM-объекты других типов, в частности, интерфейсы IOleControl, IOleControlSite и другие.

Распределенная COM

Первоначальная реализация COM может работать только на одном компьютере. DCOM (*Distributed COM*) использует LRPC (*Lightweight RPC*, облегченный RPC) для предоставления сервисов на другую машину. Сервисы DCOM можно использовать для создания защищенных распределенных приложений COM.

Инфраструктура COM

Двоичная структура COM-объекта

Как известно, описание класса, содержащее минимум одну чистую виртуальную функцию, представляет собой абстрактный класс. Напомним, что чистая виртуальная функция не имеет тела (тело определено как чистый спецификатор "=0"), вследствие чего нельзя создать представителя данного класса. Единственный полезный результат создания абстрактного класса — описание некоторой функциональности (определяемой чистыми виртуальными функциями), которой обязаны обладать все классы, производные от данного абстрактного.

Рассмотрим конкретный пример. Пусть требуется создать объект, издающий сигнал некоторой частоты. Назовем объект Beeper. Предполагается, что объекты Beeper будут выдавать короткий сигнал при помощи метода Beep, а частота сигнала будет задаваться при помощи свойства Tone. Предполагая, что в основе всей системы будет лежать наследование интерфейса, можно описать следующий класс:

```
class IBeeper {
public:
    // Метод, издающий короткий сигнал
    virtual long Beep() = 0;
    // Метод, возвращающий значение свойства
    virtual long get_Tone(short * pVal) = 0;
    // Метод, устанавливающий значение свойства
    virtual long put_Tone(short newVal) = 0;
};
```

Поскольку класс состоит только из чистых виртуальных функций, он является абстрактным, и мы назовем его интерфейсом. Таким образом мы определяем интерфейс как совокупность методов, задающих некоторую функциональность. При этом предполагается, что все методы интерфейса служат достижению какой-то определенной единой цели.

Заметим, что название класса начинается с буквы I (от *Interface*). Это общепринятая практика для наименования интерфейсов. Заметим также, что все методы возвращают тип *long*, а возвращаемое значение не имеет отношения к функциональной (семантической) нагрузке методов и предназначено для служебных целей.

Имея описание интерфейса IBeeper, мы теперь можем описать производный класс, наследующий и реализующий данный интерфейс.

В терминологии COM этот класс называется коклассом (*coclass*, класс COM) и его принято именовать с буквы C или с Co:

```
class CBeeper : public IBeeper {
public:
    // Конструктор
    CBeeper() { /* какой-то код */ }
    // Метод, издающий короткий сигнал
    long Beep() { /* какой-то код */ return 0; }
    // Метод, возвращающий значение свойства
    long get_Tone(short * pVal) { /* какой-то код */ return 0; }
    // Метод, устанавливающий значение свойства
    long put_Tone(short newVal) { /* какой-то код */ return 0; }
private:
    // данные класса и вспомогательные функции
};
```

Создавая объект кокласса, мы используем указатель на интерфейс:

```
IBeeper * Beeper = new CBeeper();
```

Через указатель на интерфейс мы можем получить от класса только ту функциональность, которая описана в интерфейсе, например:

```
Beeper->put_Tone( 1000 );
Beeper->Beep();
```

Через указатель на интерфейс невозможно получить доступ к данным класса иначе, чем через определенные интерфейсом методы — это и есть сильная инкапсуляция объекта. Тем не менее, мы можем получить доступ к открытым данным класса через переменную типа кокласса (типа CBeeper *). Однако особенностью технологии является создание двоичного, а не текстового, как у нас, компонента.

Чтобы получить двоичный компонент, мы должны скомпилировать такой программный модуль, который может создать объект заданного кокласса и вернуть указатель на интересующий нас интерфейс. В этом случае инкапсуляция будет полной, более того, мы не сможем узнать, как устроен кокласс, поскольку не будем располагать текстом его интерфейса. В нашем распоряжении останется только некий двоичный программный модуль, например, DLL. Мы можем создать несколько DLL, реализующих объект, поддерживающий интерфейс IBeeper, и даже написанных на разных языках! Самое главное заключается в том, чтобы внутренняя (двоичная) структура объекта во всех случаях была одной и той же — в этом случае объект можно будет использовать через указатель на интерфейс одинаковым образом в разных программных средах.

На рисунке 7 представлена двоичная структура объекта Beeper.

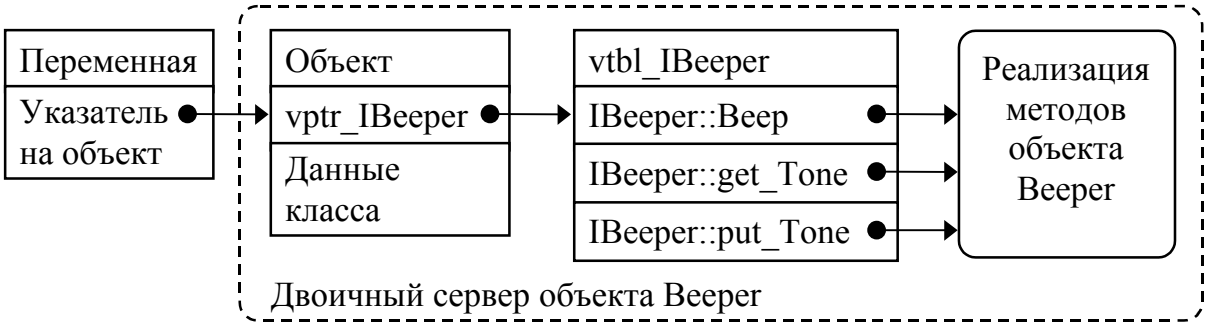


Рисунок 7 - Двоичная структура объекта Beeper

На рисунке «переменная» — это переменная Beeper (указатель на интерфейс). «Объект» — это уникальный экземпляр (instance) кокласса CBeeper. Он содержит в нашем простейшем случае указатель *vptr* на таблицу виртуальных методов *vtable* нашего интерфейса и уникальные элементы данных объекта. Таблица виртуальных методов, в свою очередь, содержит указатели на фактические реализации методов кокласса CBeeper.

В случае, когда кокласс использует множественное наследование интерфейсов, объект содержит несколько указателей vptr на несколько таблиц виртуальных методов разных интерфейсов.

Практическая работа №1

В этой практической работе мы создадим двоичный сервер DLL объекта Beeper. Это самый первый подход к рассмотрению технологии COM-объектов, который позволит нам определить проблемы, которые должна решать инфраструктура COM.

Открываем MSVS, выбираем «*create Project*», язык программирования «*Visual C++*», раздел «*Win32*», шаблон «*Win32 Project*», название проекта AFX101. Папка решения C:\AFX101. Нажимаем кнопки OK и Next. Выбираем тип проекта «DLL» и нажимаем кнопку Finish.

Открываем файл StdAfx.h и добавляем включение модуля *stdlib*:

```
#include <windows.h>
#include <stdlib.h>
```

Далее нужно создать три текстовых файла при помощи, например, программы Блокнот, и разместить их в папке проекта AFX101. Можно также просто добавить файл в проект, выбрав «*Project – Add New Item*».

Первый файл — описание интерфейса IBeeper, приведенное выше в тексте. Имя файла ibeeper.h. Кроме собственно интерфейса, файл содержит также директивы условной компиляции. Полный текст файла:

```
#if !defined(_IBEEPER_INCLUDED_)
#define _IBEEPER_INCLUDED_
```

```
class IBeeper {
public:
    // Метод, издающий короткий сигнал
    virtual long Beep() = 0;
    // Метод, возвращающий значение свойства
    virtual long get_Tone(short * pVal) = 0;
    // Метод, устанавливающий значение свойства
    virtual long put_Tone(short newVal) = 0;
};
#endif
```

Второй файл — это описание кокласса CBeeper. Имя файла, соответственно, cbeeper.h. Ниже приводится примерный текст файла:

```
#if !defined(_CBEEPER_INCLUDED_)
#define _CBEEPER_INCLUDED_
#include "ibeeper.h"
class CBeeper : public IBeeper {
public:
    // Конструктор
    CBeeper() { tone = 0; }
    // Метод, издающий короткий сигнал
    long Beep() {
        WCHAR buff[20];
        wsprintfW( buff, L"%d", tone );
        MessageBoxW(0, buff, L"Beeper", MB_OK);
        return 0;
    }
    // Метод, возвращающий значение свойства
    long get_Tone(short * pVal) {
        *pVal = tone;
        return 0;
    }
    // Метод, устанавливающий значение свойства
    long put_Tone(short newVal) {
        tone = newVal;
        return 0;
    }
private:
    // Данные класса
    short tone;
};
#endif
```

Реализация методов может быть произвольной. В приведенной реализации метод Beep выводит в окно сообщения значение частоты.

Третий файл содержит описание экспортируемых библиотекой функций. Имя файла AFX101.def. Текст этого файла:

```
LIBRARY "AFX101"
EXPORTS
    DllMain @1 PRIVATE
    DllGetBeeper @2 PRIVATE
```

Включаем новые файлы в проект при помощи меню *Project* - *Add Existing Item*.

Теперь перейдем в главный модуль проекта — AFX101.cpp. Сейчас он содержит директиву включения файла StdAfx.h. Описание функции DllMain находится в файле dllmain.cpp

Описываем директиву включения модуля cbeeper.h:

```
#include "stdafx.h"
#include "cbeeper.h"
```

Описываем функцию, которая будет создавать объект кокласса и возвращать указатель на интерфейс IBeeper:

```
VOID DllGetBeeper(void** ppv) {
    // Создаем объект Beeper
    IBeeper * Beeper = new CBeeper();
    // Возвращаем указатель на интерфейс IBeeper
    *ppv = (IBeeper *) Beeper;
}
```

Текст функции не требует особых пояснений. Компилируем проект и убеждаемся в отсутствии ошибок. Результатом компиляции является двоичный компонент AFX101.dll.

Для тестирования нашего компонента добавим новый проект. Выбираем в меню *File* - *Add* - *New Project*, «*Visual C++*», «*Win32*», «*Win32 Project*». Имя проекта Test101. Нажимаем OK и Next, выбираем переключатель «*Win32 Console Application*». Нажимаем кнопку Finish.

Проследим за тем, чтобы папка нового проекта располагалась рядом с папкой старого проекта. На вкладке *Solution Explorer* щелкнем правой кнопкой на название проекта Test101 и выберем «*Set as StartUp Project*» (сделаем этот проект стартовым).

Открываем модуль StdAfx.h нового проекта и добавляем код:

```
// TODO: reference additional headers your program requires here
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdlib.h>
```

Открываем главный модуль проекта Test101.cpp и изменяем вид функции main следующим образом:

```
void _tmain(void)
{
}
```

Перед функцией нужно добавить директиву включения файла, содержащего описание интерфейса, а также описание типа функции DLL:

```
// Описание интерфейса
#include "..\\AFX101\\ibeeper.h"
// Описание функции DLL
typedef VOID (* GETPROC)(void**);
```

Осталось выполнить следующие действия: загрузить библиотеку в память, получить указатель на функцию *DllGetBeeper*, вызвать через полученный указатель функцию и получить от нее указатель на интерфейс IBeeper созданного внутри двоичного сервера DLL объекта Beeper.

В начале функции *main* объявляем указатель на интерфейс:

```
// Указатель на интерфейс
IBeeper * Beeper = 0;
```

Далее загружаем DLL в память и получаем ее адрес:

```
// Загружаем DLL
HMODULE hmod = LoadLibrary(L"AFX101.dll");
if ( !hmod ) {
    MessageBoxW(0, L"Library not found.", L"Error", MB_OK);
    return;
}
```

Далее получаем указатель на функцию DLL:

```
// Получаем указатель на функцию DLL
GETPROC proc = (GETPROC)GetProcAddress(hmod, "DllGetBeeper");
if (!proc) {
    MessageBoxW( 0, L"Function not found.", L"Error", MB_OK );
    return;
}
```

Получив указатель, вызываем функцию DLL:

```
// Вызываем функцию DLL
(proc)((void**) & Beeper);
if (!Beeper) {
    MessageBoxW( 0, L"Object not found.", L"Error", MB_OK );
    return;
}
```

Теперь можно использовать объект Beeper через интерфейс IBeeper:

```
// Вызываем свойство
Beeper->put_Tone( 100 );
// Вызываем метод
Beeper->Beep();
```

Наконец, перед завершением функции освобождаем библиотеку:

```
// Освобождаем библиотеку
FreeLibrary( hmod );
```

Компилируем проект, чтобы убедиться в отсутствии ошибок. Запускаем проект на выполнение и убеждаемся, что объект создается внутри сервера и мы действительно можем использовать его интерфейс.

Стоит задуматься над тем, что и как мы сделали.

Во-первых, вызывая функцию *DllGetBeeper*, мы создаем объект, но не уничтожаем его впоследствии — у нас нет для этого никаких средств. Управление временем жизни объекта является одной из задач, которая должна быть решена в COM.

Другой важный вопрос — создание объекта. Мы использовали специальную функцию, чтобы создать объект. Должно быть очевидно, что для создания объектов требуется какой-то более общий механизм, нежели специальная функция для создания конкретного объекта. Вероятно, это также будет специальная функция, но она должна понимать, объект какого класса мы хотим создать, если, например, сервер может создавать объекты разных классов (что часто встречается на практике). Возникает вопрос об идентификации коклассов.

Следующий вопрос — идентификация интерфейсов объектов. Рассмотрим случай, когда два разных сервера могут создавать объекты разных коклассов и, главное, разной функциональности (семантики), описываемой, по случайному совпадению, интерфейсами с одним и тем же названием. Ведь любые два отдельно взятых программиста могут называть интерфейсы произвольно, поэтому вероятность того, что названия интерфейсов совпадут, достаточно велика.

Далее, описание интерфейса у нас выполнено на языке C++. Если мы захотим использовать объект в среде разработки, например, Delphi, нам придется переписывать интерфейс на языке ObjectPascal. Получается некоторое недоразумение. С одной стороны, сервер объекта может быть написан на любом языке программирования, но описание экспор

тируемых им интерфейсов должны соответствовать языку программирования, на котором пишется клиент данного сервера.

На все эти вопросы мы получим ответы в последующих разделах пособия. Сейчас же заметим, что эти вопросы решаются в рамках так называемой инфраструктуры COM, состоящей из библиотеки COM, которая должна присутствовать на любой платформе, поддерживающей COM, записей в реестре, специального языка для описания интерфейсов, библиотек типов и других программных компонентов. Они не являются промежуточным программным обеспечением, а лишь вспомогательными средствами, обеспечивающими правильное функционирование COM-объектов на конкретной вычислительной платформе.

GUID

GUID (Globally Unique Identifier, глобально-уникальный идентификатор) — 128-битная структура, используемая для идентификации интерфейсов, коклассов, библиотек типов, типов, перечислений и т.п.

Microsoft разработала алгоритм, генерирующий число, с большой вероятностью являющееся уникальным в пространстве и времени. Это число принято записывать в шестнадцатеричном виде с разделением на пять групп. Например, GUID важнейшего интерфейса COM IUnknown имеет вид 00000000-0000-0000-C000-000000000046. В свободной энциклопедии Интернет «Википедия» отмечается, что «генерируя 1 триллион ключей каждую наносекунду, перебрать все возможные значения удастся лишь за 10 миллиардов лет».

Структура GUID имеет следующий вид (ассемблер):

```
GUID STRUCT
    Data1    DD       ; 4 байта
    Data2    DW       ; 2 байта
    Data3    DW       ; 2 байта
    Data4    DB 8     ; 8 байт
GUID ENDS
```

В Microsoft Visual Studio 6.0 есть приложение GUIDGEN.exe, с помощью которого можно сгенерировать новый GUID и скопировать его в буфер обмена в одном из четырех предопределенных форматов, используемых в разных случаях. Сразу после запуска этого приложения генерируется новый GUID.

Программно получить новый GUID можно при помощи системной функции CoCreateGUID, единственным параметром которой является структура GUID. Существуют и другие системные функции для генера

ции уникальных идентификаторов. Можно, например, сгенерировать множество подряд идущих GUID.

На практике используются несколько разных обозначений GUID, такие, как UUID (Universally Unique Identifier, всемирно-уникальный идентификатор; под UUID понимают также стандарт идентификации, разработанный OSF — Open Software Foundation), CLSID — идентификатор класса (кокласса), IID — идентификатор интерфейса, LIBID — идентификатор библиотеки типов и др.

Первоначальная версия алгоритма Microsoft для генерации GUID использовала MAC-адрес сетевой карты компьютера, однако в настоящее время этот адрес не используется.

Интерфейс IUnknown

Класс становится классом COM тогда и только тогда, когда он поддерживает интерфейс IUnknown. Интерфейс IUnknown — базовый стандартный интерфейс, от которого производятся все остальные интерфейсы. Во всех интерфейсах COM первые три метода, таким образом, — это методы интерфейса IUnknown. Упрощенная версия этого интерфейса, определенная, например, в файле unknwn.h, выглядит так:

```
interface IUnknown {
    virtual HRESULT QueryInterface(REFIID riid, void** ppv) = 0;
    virtual ULONG AddRef(void) = 0;
    virtual ULONG Release(void) = 0;
};
```

Интерфейс IUnknown является базовым, потому что он обеспечивает базовую функциональность любого COM-объекта, а именно — возможность получения указателя на интерфейс у самого объекта при помощи метода QueryInterface, и управление временем жизни созданного объекта при помощи методов AddRef и Release.

Рассмотрим метод QueryInterface. Первый параметр типа REFIID — это уникальный идентификатор интерфейса (IID). Второй параметр — это возвращаемый указатель на запрашиваемый интерфейс.

Поскольку любой интерфейс COM наследует IUnknown, любой интерфейс имеет метод QueryInterface, что означает, что имея указатель на какой-либо интерфейс COM-объекта, клиент всегда может получить указатель на любой другой интерфейс, используя указанный метод.

Управление временем жизни объекта производится при помощи специальной внутренней переменной сервера — счетчика ссылок на объект. Каждый раз, когда клиент получает ссылку на интерфейс COM-объекта, счетчик увеличивается на единицу при помощи вызова функ

ции AddRef. Каждый раз, когда клиент освобождает интерфейс, он вызывает функцию Release, которая уменьшает счетчик на единицу и, если счетчик достиг нулевого значения, уничтожает COM-объект.

Правила использования функций AddRef и Release просты: при вызове функции QueryInterface функция AddRef вызывается автоматически в коде функции QueryInterface. Во всех других случаях (например, при копировании ссылки) клиент обязан вызывать функцию AddRef самостоятельно. Функция Release всегда вызывается клиентом. При программировании COM-объектов чрезвычайно важно следить за правильным вызовом указанных функций. Пример реализации методов интерфейса IUnknown мы увидим позднее в практической работе.

HRESULT

Все методы интерфейсов должны по возможности возвращать тип HRESULT (эквивалентный long). Связано это с двоичной сущностью объектов COM. Поскольку используемые в COM объекты сильно инкапсулированы, нет никакой возможности проконтролировать исполнение методов интерфейсов. На объекты COM возлагается дополнительная функциональная обязанность, кроме той, которая предусматривается собственно интерфейсом — сообщить клиенту об успешности выполнения метода. Поэтому все типы, используемые методом для выполнения прямой функциональности, передаются объекту как аргументы, а возвращаемое значение типа HRESULT используется для сигнализации результата выполнения метода. Можно сказать, что возвращаемое значение метода интерфейса сообщает о результате исполнении кода метода, но не о результате вычисления. Результат вычисления, если он есть (что справедливо для методов, являющихся функциями, и методов, возвращающих значения свойств), в COM всегда является последним аргументом метода и помечается в библиотеке типов особым образом.

HRESULT — это 32-битное поле, приведенное на рисунке 8.

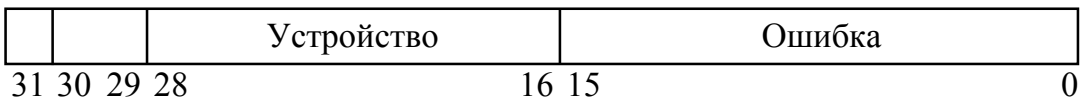


Рисунок 8 - HRESULT

Старший бит используется для определения успешности вызова. Биты 30-29 зарезервированы. Биты 28-16 указывают на устройство, вызвавшее ошибку. Биты 15-0 содержат информацию об ошибке.

Методы COM должны возвращать 0 в случае успешности вызова. Для этой цели в MSVS определена константа S_OK. Для информации о наиболее распространенных ошибках используются константы:

HRESULT	Значение
E_UNEXPECTED	Катастрофическая ошибка
E_NOTIMPL	Метод не имеет реализации
E_OUTOFMEMORY	Недостаточно памяти
E_INVALIDARG	Неверные аргументы или неправильный вызов
E_NOINTERFACE	Неподдерживаемый интерфейс
E_POINTER	Недопустимый указатель (равен нулю)
E_HANDLE	Недопустимый идентификатор
E_ABORT	Операция прервана
E_FAIL	Что-то не сработало

Для проверки возвращаемого значения в COM определены два макроопределения. Макрос FAILED возвращает истину, если параметр содержит ошибку (установленный старший бит). Макрос SUCCEEDED, наоборот, возвращает в этом случае ложь.

Пример использования макроса SUCCEEDED:

```
HRESULT hr = вызов_COM_метода();
if (SUCCEEDED(hr)) {
    return S_OK;
} else {
    return E_FAIL;
}
```

Описание интерфейсов с помощью макросов COM

Для описания методов интерфейсов COM в MSVS используются стандартные макроопределения STDMETHOD и STDMETHOD_:

```
#define STDMETHOD(method) virtual HRESULT STDMETHODCALLTYPE method
#define STDMETHOD_(type, method) virtual type STDMETHODCALLTYPE method
```

Здесь STDMETHODCALLTYPE — это макрос, определяющий способ вызова, который в WIN32 преобразуется в __stdcall.

Первый из макросов STDMETHOD определяет метод с возвращаемым значением HRESULT, а второй — с произвольным возвращаемым значением type.

С использованием данных макросов описание интерфейса IBeeper примет следующий вид:

```
interface IBeeper : public IUnknown {
    STDMETHOD(Beep)() = 0;
    STDMETHOD(get_Tone)(short * pVal) = 0;
    STDMETHOD(put_Tone)(short newVal) = 0;
};
```

Для описания метода как чистого можно также использовать макрос PURE. Кроме того, для описания собственно интерфейса используется макрос DECLARE_INTERFACE_:

```
DECLARE_INTERFACE_(IBeeper, public IUnknown)
{
    STDMETHOD(Beep)() PURE;
    STDMETHOD(get_Tone)(short * pVal) PURE;
    STDMETHOD(put_Tone)(short newVal) PURE;
};
```

Это описание полностью аналогично приведенному выше.

Для определения методов интерфейса при описании кокласса и его реализации используются аналогичные макросы, но с IMPL на конце.

Практическая работа №2

В рамках данной работы мы создадим настоящий COM-объект, наследующий настоящий COM интерфейс IUnknown.

Открываем MSVS, выбираем «*create Project*», язык программирования «*Visual C++*», раздел «*Win32*», шаблон «*Win32 Project*», название проекта AFX102. Папка решения C:\AFX102. Нажимаем кнопки OK и Next. Далее выбираем тип проекта «DLL» и нажимаем кнопку Finish.

Открываем файл StdAfx.h и добавляем включение модулей, необходимых для реализации сервера COM-объекта:

```
#include <windows.h>
#include <stdlib.h>
#include <objbase.h>
#include <initguid.h>
```

Далее нужно создать четыре текстовых файла при помощи, например, программы Блокнот, и разместить их в папке C:\AFX102.

Первый файл — это описание константы, идентифицирующей наш интерфейс IBeeper. Имя файла iid.h. Для генерации GUID интерфейса запустите приложение GUIDGEN.exe, скопируйте новый GUID в формате 2 и вставьте в файл iid.h. Должно получиться примерно следующее:

```
// iid.h
#if !defined(_IID_INCLUDED_)
#define _IID_INCLUDED_
// {758929A2-270F-4fad-9B9D-B7585164C3FD}
DEFINE_GUID(IID_IBeeper,
0x758929a2,0x270f,0x4fad,0x9b,0x9d,0xb7,0x58,0x51,0x64,0xc3,0xfd);
#endif
```

Заметим, что фактический GUID у вас должен другим.

Второй файл — новое описание интерфейса IBeeper, приведенное выше в тексте. Имя файла ibeeper.h. Текст файла имеет следующий вид:

```
// ibeeper.h
#if !defined(_IBEEPER_INCLUDED_)
#define _IBEEPER_INCLUDED_
DECLARE_INTERFACE_(IBeeper, IUnknown)
{
    // Метод, издающий короткий сигнал
    STDMETHOD(Beep)() PURE;
    // Метод, возвращающий значение свойства
    STDMETHOD(get_Tone)(short * pVal) PURE;
    // Метод, устанавливающий значение свойства
    STDMETHOD(put_Tone)(short newVal) PURE;
};
#endif
```

Третий файл — это описание кокласса CBeeper. Имя файла, соответственно, cbeeper.h. Ниже приводится примерный текст файла:

```
// cbeeper.h
#if !defined(_CBEEPER_INCLUDED_)
#define _CBEEPER_INCLUDED_
#include "ibeeper.h"
#include "iid.h"
class CBeeper : public IBeeper {
public:
    // Конструктор
    CBeeper() {
        tone = 0;
        m_refCount = 0;
    }
    /****** Интерфейс IUnknown ******/
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
    STDMETHODIMP_(ULONG)AddRef();
    STDMETHODIMP_(ULONG)Release();
    /****** Интерфейс IBeeper ******/
    // Метод, издающий короткий сигнал
    STDMETHODIMP Beep();
    // Метод, возвращающий значение свойства
    STDMETHODIMP get_Tone(short * pVal);
    // Метод, устанавливающий значение свойства
    STDMETHODIMP put_Tone(short newVal);
private:
    // счетчик ссылок
    ULONG m_refCount;
```

```
    // данные класса
    short tone;
};
/****** Реализация интерфейса IUnknown ******/
STDMETHODIMP_(ULONG)CBeeper::AddRef()
{
    return ++m_refCount;
}
STDMETHODIMP_(ULONG)CBeeper::Release()
{
    if (--m_refCount == 0) delete this;
    return m_refCount;
}
STDMETHODIMP CBeeper::QueryInterface(REFIID riid, void** ppv)
{
    if (riid == IID_IUnknown)
        *ppv = (IUnknown*)this;
    else if (riid == IID_IBeeper)
        *ppv = (IBeeper*)this;
    else {
        *ppv = 0;
        return E_NOINTERFACE;
    }
    ((IUnknown*)(*ppv))->AddRef();
    return S_OK;
}
/****** Реализация интерфейса IBeeper ******/
STDMETHODIMP CBeeper::Beep()
{
    WCHAR buff[20];
    wsprintfW( buff, L"%d", tone );
    MessageBoxW(0, buff, L"Beeper", MB_OK);
    return S_OK;
}
STDMETHODIMP CBeeper::get_Tone(short * pVal)
{
    *pVal =  tone;
    return S_OK;
}
STDMETHODIMP CBeeper::put_Tone(short newVal)
{
    tone = newVal;
    return S_OK;
}
#endif
```

Четвертый файл содержит описание экспортируемых библиотекой функций. Имя файла AFX102.def. Текст этого файла:

```
LIBRARY "AFX102"
EXPORTS
    DllMain @1 PRIVATE
    DllGetBeeper @2 PRIVATE
```

Включаем новые файлы в проект при помощи меню *Project* - *Add Existing Item*.

Теперь перейдем в главный модуль проекта — AFX102.cpp. Сейчас он содержит директиву включения файла StdAfx.h. Описание функции *DllMain* находится в файле dllmain.cpp.

Описываем директиву включения модуля cbeeper.h:

```
#include "stdafx.h"
#include "cbeeper.h"
```

Описываем функцию, которая будет создавать объект кокласса и возвращать указатель на интерфейс IUnknown:

```
VOID DllGetBeeper(void** ppv) {
    // Создаем объект Beeper
    IBeeper * Beeper = new CBeeper();
    // Возвращаем указатель на интерфейс IUnknown
    Beeper->QueryInterface( IID_IBeeper, (void**) ppv );
}
```

Текст функции не требует особых пояснений. Компилируем проект и убеждаемся в отсутствии ошибок. Результатом компиляции является двоичный компонент AFX102.dll.

Для тестирования нашего компонента добавим новый проект. Выбираем в меню *File* - *Add* - *New Project*, «*Visual C++*», «*Win32*», «*Win32 Project*». Имя проекта Test102. Нажимаем OK и Next, выбираем переключатель «*Win32 Console Application*». Нажимаем кнопку Finish.

Проследим за тем, чтобы папка нового проекта располагалась рядом с папкой старого проекта. На вкладке *Solution Explorer* щелкнем правой кнопкой на название проекта Test102 и выберем *«Set as StartUp Project*» (сделаем этот проект стартовым).

Открываем модуль StdAfx.h нового проекта и добавляем код:

```
// TODO: reference additional headers your program requires here
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdlib.h>
#include <objbase.h>
#include <initguid.h>
```

Открываем главный модуль проекта Test102.cpp и изменяем вид функции *main* следующим образом:

```
void _tmain(void)
{
}
```

Перед функцией нужно добавить директивы включения требуемых файлов, а также описание типа функции DLL:

```
// Описание интерфейса
#include "..\\AFX102\\ibeeper.h"
#include "..\\AFX102\\iid.h"
// Описание функции DLL
typedef VOID (*GETPROC)(void**);
```

Осталось выполнить следующие действия: загрузить библиотеку в память, получить указатель на функцию *DllGetBeeper*, вызвать через полученный указатель функцию и получить от нее указатель на интерфейс IUnknown созданного внутри двоичного сервера DLL объекта Beeper.

В начале функции main объявляем указатели на интерфейсы:

```
// Указатели на интерфейсы
IUnknown * pUnk = 0;
IBeeper * Beeper = 0;
```

Далее загружаем DLL в память и получаем ее адрес:

```
// Загружаем DLL
HMODULE hmod = LoadLibrary(L"AFX102.dll");
if ( !hmod ) {
    MessageBoxW( 0, L"Library not found.", L"Error", MB_OK );
    return;
}
```

Далее получаем указатель на функцию DLL:

```
// Получаем указатель на функцию DLL
GETPROC proc = (GETPROC)GetProcAddress(hmod, "DllGetBeeper");
if ( !proc ) {
    MessageBoxW( 0, L"Function not found.", L"Error", MB_OK );
    return;
}
```

Получив указатель, вызываем функцию DLL:

```
// Вызываем функцию DLL
(proc)( (void**) & pUnk );
// Проверяем результат
```

```
if ( !pUnk ) {
    MessageBoxW( 0, L"IUnknown not found.", L"Error", MB_OK );
    return;
}
```

В результате мы получаем указатель на интерфейс IUnknown. С его помощью мы теперь получим указатель на интерфейс IBeeper:

```
// Получаем интерфейс IBeeper
HRESULT hr = pUnk->QueryInterface(IID_IBeeper, (void**)&Beeper );
```

Проверяем результат вызова метода QueryInterface:

```
// Проверяем результат вызова метода COM
if ( FAILED( hr ) ) {
    MessageBoxW( 0, L"IBeeper not found.", L"Error", MB_OK );
    goto cleanup;
}
```

Здесь используется метка cleanup, которая будет описана в тексте позже. Теперь же используем объект Beeper через интерфейс IBeeper:

```
// Вызываем свойство
Beeper->put_Tone( 100 );
// Вызываем метод
Beeper->Beep();
```

Наконец, описываем метку cleanup и очистку:

```
cleanup:
    // Освобождаем указатели на интерфейсы
    if ( pUnk ) pUnk->Release();
    if ( Beeper ) Beeper->Release();
    // Освобождаем библиотеку
    FreeLibrary( hmod );
```

Тестовый проект готов. Компилируем его и убеждаемся в отсутствии ошибок. Запускаем проект на исполнение и убеждаемся, что указатели на интерфейсы реализуются должным образом.

Итак, мы создали настоящий COM-объект, наследующий интерфейс IUnknown, однако проблема создания COM-объекта по-прежнему осталась нерешенной. В последующих разделах мы продолжим изучение инфраструктуры COM и рассмотрим, как устроен сервер COM и как создаются COM-объекты внутри этого сервера.

Фабрика класса

Объекты COM обычно находятся внутри некоторого сервера, в качестве которого выступает файл типа DLL, EXE или OCX. Создать объ

ект, который находится внутри двоичного файла, можно только средствами самого этого файла (учитывая также, что сервер может быть написан на произвольном языке программирования). Принцип прозрачности COM (transparency) требует, чтобы объекты создавались унифицированным способом. Именно для этой цели служит стандартный интерфейс COM IClassFactory. Его упрощенное определение, приведенное в файле unknwn.h, выглядит примерно следующим образом:

```
interface IClassFactory : public IUnknown {
    virtual HRESULT CreateInstance(LPUNKNOWN pUnkOuter,
        REFIID riid,
        void** ppv
    ) = 0;
    virtual HRESULT LockServer(BOOL fLock) = 0;
};
```

Метод CreateInstance предназначен для создания объекта кокласса. Параметрами метода являются:

- ссылка на интерфейс IUnknown агрегирующего объекта (если таковой есть), про агрегацию см. далее раздел «Повторное использование объектов»;
- идентификатор запрашиваемого интерфейса;
- возвращаемый указатель на интерфейс.

Метод LockServer блокирует сервер в памяти, предотвращая его выгрузку в некоторых случаях, обычно при необходимости создать множество COM-объектов.

Заметим, что существует также интерфейс IClassFactory2, с помощью которого создаются лицензируемые объекты. Рассмотрение этого интерфейса выходит за рамки данного пособия.

Объект класса (*class object*) — это объект COM, реализующий интерфейс IClassFactory. Объекты класса называют также фабриками класса (*class factory*). Объект класса предназначен только для того, чтобы создавать объекты другого конкретного класса (кокласса).

Может возникнуть небольшая путаница с терминологией. Под объектом обычно понимается представитель класса (*instance*). Здесь же словосочетание «объект класса» получает другой смысл — фабрика класса.

Для каждого COM-объекта, базирующегося внутри сервера, существует отдельная фабрика класса. Иначе говоря, фабрика класса всегда создает объекты строго определенного класса и является, таким образом, неотъемлемой частью COM-объекта. Рисунок 9 поясняет соотношение между COM-объектами и объектами класса.

Таким образом, чтобы создать представителя класса, сначала нужно создать фабрику этого класса. Кто же создает фабрику класса?

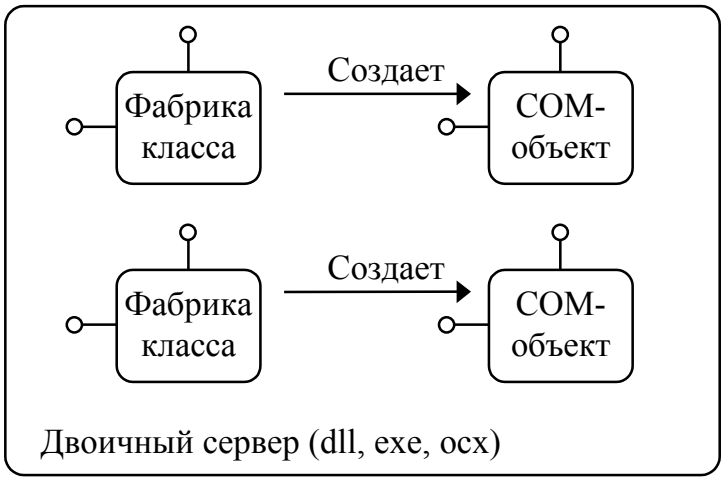


Рисунок 9 - COM-объекты и фабрики класса

Функции COM DLL

Как уже было сказано, объекты COM базируются внутри некоторого двоичного сервера. В качестве сервера могут выступать файлы типа DLL, EXE или OCX.

Различия между файлами типа DLL и OCX несущественны, поэтому эти типы серверов можно рассматривать как одинаковые. Серверы типа EXE (называемые иногда сервисами) являются достаточно специфичными, и подробное их рассмотрение выходит за рамки настоящего пособия. Поэтому сейчас мы сосредоточимся на серверах типа DLL.

Любая DLL на основе COM выполняет свои обязанности через экспорт следующих четырех функций:

1. *DllCanUnloadNow*. Эта функция отвечает на вопрос: «Можно ли выгрузить DLL из памяти?». Фактически она управляет временем жизни сервера, и работает по тому же принципу, что и функции, управляющие временем жизни объекта кокласса. Для ее реализации используется глобальный счетчик используемых объектов сервера, который изменяется в конструкторе и деструкторе каждого экспортируемого кокласса.

2. *DllGetClassObject*. Возвращает объект класса (фабрику класса) определенного кокласса. Функция имеет три параметра:

- идентификатор кокласса CLSID;
- идентификатор запрашиваемого интерфейса; как правило, это интерфейс IClassFactory;
- возвращаемый указатель на запрашиваемый интерфейс.

Клиент, получив указатель на интерфейс, например, IClassFactory, далее может создать представителя кокласса, при необходимости заблокировав сервер в памяти на время создания множества объектов. Заме

тим, что для каждого кокласса должен быть сгенерирован соответствующий GUID под названием CLSID.

Для получения объекта класса библиотека COM определяет функцию *CoGetClassObject*, которую может использовать клиент. Эта функция, при наличии в сервере запрашиваемого кокласса, загружает сервер в память при помощи функции *CoLoadLibrary*, а уже эта функция вызывает функцию сервера *DllGetClassObject*.

3. *DllRegisterServer*. С помощью этой функции сервер вносит специальные записи в реестр операционной системы (регистрируется).

4. *DllUnregisterServer*. С помощью этой функции сервер удаляет записи из реестра операционной системы (дерегистрируется).

Очевидно, что следующий вопрос, который нам нужно разобрать — это регистрация серверов COM в реестре операционной системы.

COM записи в реестре

Зачем нужна регистрация сервера? Вспомним наш практический опыт, полученный в предыдущих работах. Чтобы создать представителя кокласса, мы должны загрузить в память соответствующий сервер. Сервер — это двоичный файл, расположенный где-то внутри файловой системы компьютера. Как найти его?

Все очень просто. В реестр операционной системы записывается соотношение двух строк: CLSID — путь к серверу. Зная CLSID интересующего нас кокласса, из реестра мы можем получить путь к серверу, загрузить его в память и получить фабрику класса.

На самом деле в реестр записывается не одно, а минимум три важных соотношения:

1. ProgID — CLSID;
2. CLSID — ProgID;
3. CLSID — путь_к_серверу.

Первые два соотношения позволяют узнать идентификатор кокласса по его программному идентификатору и наоборот, а последнее, как уже было сказано, — найти сервер по идентификатору кокласса. Поиск идентификаторов и путей к серверам выполняет специальное программное обеспечение — библиотека COM, реализованная как набор системных функций. Функции библиотеки COM начинаются с префикса Co.

Новый вопрос — что такое программный идентификатор кокласса ProgID (Programmatic Identifier)?

Для создания кокласса программист, естественно, использует строковый идентификатор, например, CBeeper. Учитывая, что кокласс располагается внутри сервера (у которого также есть имя), получается уни

кальное сочетание — ИмяСервера и ИмяКокласса, которые соединяются точкой с получением программного идентификатора ProgID.

На самом деле полный ProgID состоит из трех частей, соединяемых точками: ИмяСервера, ИмяКокасса и номер версии кокласса. Таким образом, ProgID = ИмяСервера.ИмяКокласса.Версия.

Программный идентификатор используется языками, не поддерживающими принципы классического объектно-ориентированного программирования, такими, например, как Visual Basic и скрипты, в рамках программного управления серверами с использованием технологии OLE Automation.

Перейдем теперь к рассмотрению собственно реестра Windows.

Реестр операционной системы Windows — это специальная системная база данных для хранения разнообразной информации о системе, пользователях и приложениях. Здесь мы рассматриваем реестр только с точки зрения функционирования COM-объектов.

Реестр разбит на несколько узлов, называемых ульями (hives). Для разработчика COM важнейшим является узел HKEY_CLASSES_ROOT, сокращенно HKCR. Точки входа под ульями называются ключами, они могут содержать вложенные ключи.

Для просмотра и редактирования реестра в Windows есть приложение regedit.exe, которое используется здесь для демонстрации содержимого реестра. Чтобы открыть это приложение, нажмите кнопку Пуск, выберите «Выполнить», введите «regedit» и нажмите OK. Заметим, что редактирование реестра с помощью приложения regedit может привести к катастрофическим последствиям (краху системы).

Первое, что содержится под узлом HKCR, — это список расширений, зарегистрированных в системе. Далее следуют программные идентификаторы ProgID. На рисунке 10 в качестве примера приведен программный идентификатор *Access.Application*. Учитывая, что на рисунке раскрыт ключ CLSID, видно, какой CLSID соответствует этому программному идентификатору. Таким образом, на рисунке 10 мы видим соотношение ProgID — CLSID.

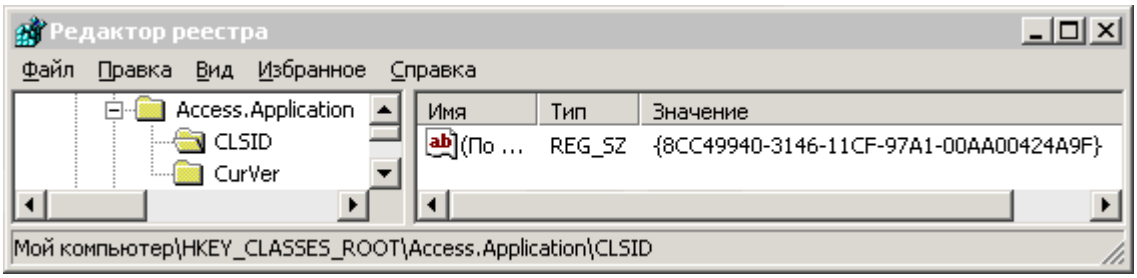


Рисунок 10 - ProgID

Далее в узле HKCR есть ключ CLSID, который содержит информацию о серверах COM. На рисунке 11 приведен ключ раздела CLSID, соответствующий ProgID *Access.Application*, приведенному на рисунке 10.

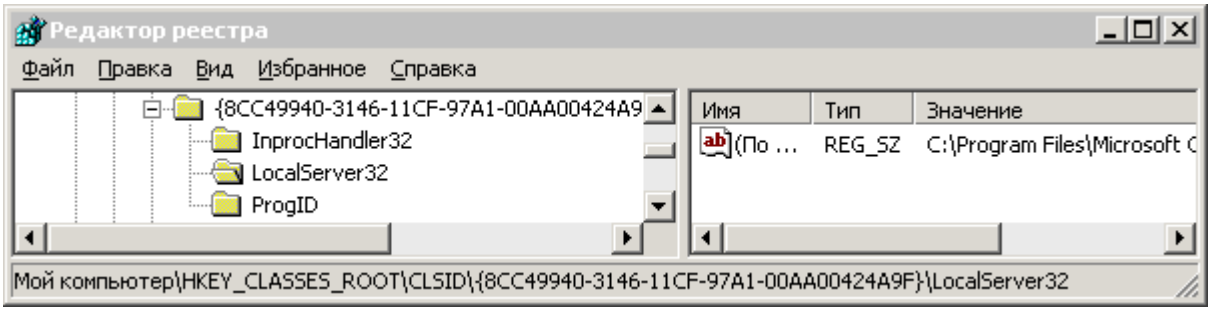


Рисунок 11 - CLSID

Учитывая, что на рисунке раскрыт ключ LocalServer32, мы можем проследить путь к соответствующему серверу в правой части редактора реестра. Ключи конкретного CLSID могут содержать следующие ключи:

- ProgID — программный идентификатор кокласса;
- VersionIndependentProgID — ProgID без номера версии;
- InprocServer32 — путь к серверу типа DLL или OCX;
- LocalServer32 — путь к серверу типа EXE;
- другие ключи.

Таким образом, на рисунке 11 мы видим, как в реестре записываются соотношения CLSID — ProgID и CLSID — путь_к_серверу.

Заметим, что реестр содержит также и другую информацию о COM, например, записи об интерфейсах.

Регистрация сервера

Как мы выяснили в предыдущих разделах, для правильного функционирования COM-объектов каждый сервер должен быть зарегистрирован в реестре операционной системы. Здесь мы рассмотрим, как это может быть сделано.

Во многих практических случаях у нас есть некоторый сервер в виде двоичного файла типа DLL или EXE, и нам нужно зарегистрировать этот сервер. Для регистрации сервера DLL в этом случае нужно использовать утилиту операционной системы — программу regsvr32.exe.

Эта утилита используется в командной строке. Нужно ввести название утилиты regsvr32 и путь к серверу, например:

```
regsvr32 c:\AFX103\Debug\AFX103.dll
```

Утилита regsvr32 вызывает функцию сервера DllRegisterServer и сообщает о результате ее выполнения при помощи стандартного диалогового окна, примерный вид которого приведен на рисунке 12.

Рисунок 12 - Регистрация сервера при помощи regsvr32

Иногда встречается ситуация, когда нужно удалить информацию о сервере DLL из реестра. В этом случае нужно использовать ключ /u. Например, для дерегистрации сервера AFX103.dll из предыдущего примера нужно ввести командную строку:

```
regsvr32 c:\AFX103\Debug\AFX103.dll /u
```

Утилита regsvr32 вызывает в этом случае другую функцию сервера DLL — DllUnregisterServer. При выполнении этого действия утилита regsvr32 также сообщает о результате при помощи стандартного диалогового окна, приведенного на рисунке 13.

Рисунок 13 - Дерегистрация сервера при помощи regsvr32

Таким образом, серверы DLL являются саморегистрирующимися.

Заметим, что в обоих случаях использование ключа /s (от silent) позволяет избежать вывода стандартного сообщения. Это особенно полезно для создания пакетного файла регистрации (или дерегистрации).

Серверы типа EXE регистрируются и дерегистрируются иначе, поскольку они являются исполняемыми программами. Для регистрации сервера EXE в командной строке нужно ввести путь к серверу с ключом /regserver, а для дерегистрации — с ключом /unregserver.

Зарегистрировать сервер типа DLL можно также при помощи специального файла типа .reg, который содержит отношения, записываемые в реестр. Первая строка файла должна содержать REGEDIT, последующие строки — записываемые отношения, по одному в строке. Например, для регистрации сервера AFX103.dll, который будет создан нами в очередной практической работе, можно использовать следующий текстовый файл (имя файла AFX103.reg):

```
REGEDIT
HKEY_CLASSES_ROOT\AFX103.Beeper\CLSID = {3C66CCE7-95A1-4f47-A8F3-2B1A975C8A2E}
HKEY_CLASSES_ROOT\CLSID\{3C66CCE7-95A1-4f47-A8F3-2B1A975C8A2E} = AFX103.Beeper
HKEY_CLASSES_ROOT\CLSID\{3C66CCE7-95A1-4f47-A8F3-2B1A975C8A2E}\InprocServer32 =
C:\AFX103\Debug\AFX103.dll
```

Заметим, что файл состоит из четырех строк (последняя строка просто не умещается на странице документа), а знаки "=" должны быть окружены в точности одним пробелом с каждой стороны. *Убедитесь в правильном пути к вашему серверу*.

Кроме того, автоматическая регистрация сервера производится при его создании средствами *Microsoft Visual Studio*.

Библиотека COM

На вычислительной системе, поддерживающей COM, должна присутствовать библиотека COM, состоящая из набора функций. Каждая функция библиотеки COM начинается с префикса Co. В этом разделе рассматриваются только некоторые из функций.

CoInitialize — инициализирует библиотеку COM для текущего потока с использованием однопоточной модели апартаментов. Единственный параметр должен быть равен NULL. Эта функция должна вызываться первой среди других функций библиотеки COM. Для многопоточной модели следует использовать функцию CoInitializeEx.

CoUninitialize — закрывает библиотеку COM для текущего потока.

CoCreateInstance — создает одиночный COM-объект на локальной машине. Для создания одиночного COM-объекта на удаленной машине следует использовать функцию CoCreateInstanceEx.

У функции CoCreateInstance пять параметров:

1) идентификатор кокласса;

2) указатель на IUnknown агрегирующего объекта;

3) контекст;

4) идентификатор запрашиваемого интерфейса;

5) возвращаемый указатель на запрашиваемый интерфейс.

Контекст определяется константой перечисления CLSCTX. Наиболее часто используются константы:

CLSCTX_INPROC_SERVER — создание объекта в процессе клиента (сервер DLL);

CLSCTX_LOCAL_SERVER — создание объекта вне процесса клиента (сервер EXE);

CoGetClassObject — создает фабрику класса для заданного CLSID.

Параметры функции:

1) идентификатор кокласса;

2) контекст;
3) указатель на машину, на которой создается фабрика класса;
4) идентификатор запрашиваемого интерфейса;
5) возвращаемый указатель на запрашиваемый интерфейс.
Следующие две функции не являются частью библиотеки COM.
CLSIDFromProgID — возвращает CLSID по заданному ProgID.
Параметры функции:
1) ProgID (тип LPOLESTR);
2) возвращаемый ProgID (тип LPCLSID).
ProgIDFromCLSID — возвращает ProgID по заданному CLSID.
Параметры функции:
1) CLSID (тип REFCLSID);
2) указатель на возвращаемый ProgID (тип LPOLESTR).

Практическая работа №3

В этой работе мы рассмотрим создание объекта Beeper с использованием фабрики класса. Сервер DLL объекта практически полностью аналогичен серверу из предыдущей работы, поэтому можно использовать часть файлов из предыдущей работы.

Открываем MSVS, выбираем «*create Project*», язык программирования «*Visual C++*», раздел «*Win32*», шаблон «*Win32 Project*», название проекта AFX103. Папка решения C:\AFX103. Нажимаем кнопки OK и Next. Далее выбираем тип проекта «DLL» и нажимаем кнопку Finish.

Открываем файл StdAfx.h и добавляем включение модулей, необходимых для реализации сервера COM-объекта:

```
#include <windows.h>
#include <stdlib.h>
#include <objbase.h>
#include <initguid.h>
```

Скопируем в папку проекта файлы iid.h, ibeeper.h, cbeeper.h, созданные в ходе выполнения предыдущей работы. Включим указанные файлы в проект при помощи *Project - Add Existing Item*.

При помощи приложения GUIDGEN.exe сгенерируем новый GUID для кокласса CBeeper и скопируем его в форме 2 в буфер обмена. Добавляем новый CLSID в файл iid.h, при этом должно получиться примерно следующее (GUID у вас должен быть другим):

```
// iid.h
. . .
// {3C66CCE7-95A1-4f47-A8F3-2B1A975C8A2E}
DEFINE_GUID(CLSID_Beeper,
0x3c66cce7,0x95a1,0x4f47,0xa8,0xf3,0x2b,0x1a,0x97,0x5c,0x8a,0x2e);
```

Для описания фабрики класса создадим текстовый файл clsfct.h. Примерный код этого файла приведен ниже:

```
// clsfct.h
#ifndef _CLSFCT_H_INCLUDED_
#define _CLSFCT_H_INCLUDED_
#include <windows.h>
extern ULONG g_objCount;
extern ULONG g_lockCount;
// Фабрика класса
class BeeperClassFactory : public IClassFactory {
public:
    BeeperClassFactory()
    {
        m_refCount = 0;
        ++g_objCount;
    }
    ~BeeperClassFactory()
    {
        --g_objCount;
    }
    /****** Интерфейс IUnknown ******/
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
    STDMETHODIMP_(ULONG)AddRef();
    STDMETHODIMP_(ULONG)Release();
    /****** Интерфейс IClassFactory ******/
    STDMETHODIMP LockServer(BOOL fLock);
    STDMETHODIMP CreateInstance(LPUNKNOWN pUnkOuter, REFIID,
void**);
private:
    ULONG m_refCount;
};
/****** Интерфейс IUnknown ******/
STDMETHODIMP_(ULONG)BeeperClassFactory::AddRef()
{
    return ++m_refCount;
}
STDMETHODIMP_(ULONG)BeeperClassFactory::Release()
{
    if (--m_refCount == 0) delete this;
    return m_refCount;
}
STDMETHODIMP BeeperClassFactory::QueryInterface(REFIID riid, void **
ppv)
```

```
{
    if ( riid == IID_IUnknown )
        *ppv = (IUnknown*)this;
    else if ( riid == IID_IClassFactory )
        *ppv = (IClassFactory*)this;
    else {
        *ppv = 0;
        return E_NOINTERFACE;
    }
    ((IUnknown*)(*ppv))->AddRef();
    return S_OK;
}
/****** Интерфейс IClassFactory ******/
STDMETHODIMP BeeperClassFactory::CreateInstance(LPUNKNOWN pUnkOuter,
REFIID riid, void** ppv)
{
    // агрегация не поддерживается
    if ( pUnkOuter != 0 ) return CLASS_E_NOAGGREGATION;
    CBeeper * pBeeper = 0;
    // создаем объект Beeper
    pBeeper = new CBeeper;
    // запрашиваем интерфейс
    HRESULT hr = pBeeper->QueryInterface( riid, ppv );
    if ( FAILED( hr ) ) delete pBeeper;
    return hr;
}
STDMETHODIMP BeeperClassFactory::LockServer(BOOL fLock)
{
    if (fLock)
        ++g_lockCount;
    else
        --g_lockCount;
    return S_OK;
}
#endif
```

Включим файл в проект.

Перейдем к реализации функций сервера. Файл AFX103.cpp. В начале этого файла включаем в него необходимые модули:

```
// AFX103.cpp
#include "stdafx.h"
#include "iid.h"
#include "cbeeper.h"
```

```
#include "clsfct.h"
// Глобальные счетчики
ULONG g_objCount = 0;
ULONG g_lockCount = 0;
```

Далее описываем две функции DLL.

```
STDAPI DllCanUnloadNow()
{
    if (g_lockCount == 0 && g_objCount == 0) return S_OK;
    return S_FALSE;
}
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, void** ppv)
{
    // Указатель на фабрику класса Beeper
    BeeperClassFactory *pFact = 0;
    // этот сервер создает только Beeper
    if ( rclsid != CLSID_Beeper ) return CLASS_E_CLASSNOTAVAILABLE;
    // создаем фабрику класса Beeper
    pFact = new BeeperClassFactory;
    // получаем интерфейс (видимо, IClassFactory)
    HRESULT hr = pFact->QueryInterface( riid, (void**)ppv );
    if ( FAILED( hr ) ) delete pFact;
    return hr;
}
```

Наконец, чтобы сделать функции сервера видимыми окружающему миру, добавим файл AFX103.def для их определения:

```
LIBRARY "AFX103"
EXPORTS
    DllMain @1 PRIVATE
    DllCanUnloadNow @2 PRIVATE
    DllGetClassObject @3 PRIVATE
```

Построим проект и устраним ошибки, если они есть.

Для тестирования нашего компонента добавим новый проект. Выбираем в меню *File - Add - New Project*, «*Visual C++*», «*Win32*», «*Win32 Project*». Имя проекта Test103. Нажимаем OK и Next, выбираем переключатель «*Win32 Console Application*». Нажимаем кнопку Finish.

Проследим за тем, чтобы папка нового проекта располагалась рядом с папкой старого проекта. На вкладке *Solution Explorer* щелкнем правой кнопкой на название проекта Test103 и выберем *«Set as StartUp Project*» (сделаем этот проект стартовым).

Открываем модуль StdAfx.h нового проекта и добавляем код:

```
// TODO: reference additional headers your program requires here
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdlib.h>
#include <objbase.h>
#include <initguid.h>
```

Открываем модуль Test103.cpp. Изменяем вид функции *main*, добавляем перед ней две вспомогательные функции и директивы включения необходимых файлов:

```
// Описание интерфейса
#include "..\\AFX103\\ibeeper.h"
#include "..\\AFX103\\iid.h"

// CoGetClassObject
void main_a(void) {
}

// CoCreateInstance
void main_b(void) {
}

// Основная функция
void _tmain(void) {
    main_a();
    main_b();
}
```

В функции *main_a* сначала получим объект класса, а затем с его помощью получим указатель на интерфейс IBeeper:

```
void main_a(void) {
    // Указатели на интерфейсы
    IClassFactory *pFact = 0;
    IBeeper *Beeper = 0;
    // инициализируем библиотеку COM
    CoInitialize(0);
    // получаем указатель на фабрику класса
    HRESULT hr = CoGetClassObject(CLSID_Beeper,
        CLSCTX_INPROC_SERVER, 0, IID_IClassFactory, (void**)&pFact);
    if ( FAILED( hr ) ) {
        MessageBoxW(0, L"IClassFactory not found.", L"Error", MB_OK);
        goto cleanup;
    }
    hr = pFact->CreateInstance(0, IID_IBeeper, (void**) & Beeper);
```

```
    if ( FAILED( hr ) ) {
        MessageBoxW(0, L"IBeeper not found.", L"Error", MB_OK);
        goto cleanup;
    }
    // Вызываем свойство
    Beeper->put_Tone( 50 );
    // Вызываем метод
    Beeper->Beep();
cleanup:
    if (pFact) pFact->Release();
    if (Beeper) Beeper->Release();
    CoUninitialize();
}
```

Компилируем и запускаем проект на выполнение. Убеждаемся, что создать интерфейс IClassFactory невозможно. Дело в том, что мы еще не зарегистрировали сервер. Сделать это можно при помощи, например, файла AFX103.reg, упоминавшегося ранее. После регистрации реализация интерфейсов DLL должна пройти успешно.

В функции *main_b* мы пойдем более коротким путем, а именно — используя функцию *CoCreateInstance*:

```
void main_b(void) {
    // Указатель на интерфейс
    IBeeper *Beeper = 0;
    // инициализируем библиотеку COM
    CoInitialize(0);
    // сразу получаем указатель на интерфейс
    HRESULT hr = CoCreateInstance(CLSID_Beeper, 0,
        CLSCTX_INPROC_SERVER, IID_IBeeper, (void**) & Beeper);
    if ( FAILED( hr ) ) {
        MessageBoxW(0, L"IBeeper not found.", L"Error", MB_OK);
        goto cleanup;
    }
    // Вызываем свойство
    Beeper->put_Tone( 100 );
    // Вызываем метод
    Beeper->Beep();
cleanup:
    if (Beeper) Beeper->Release();
    CoUninitialize();
}
```

Еще раз компилируем и тестируем проект, убеждаемся, что интерфейсы сервера реализуются должным образом.

Таким образом, мы создали сервер COM-объекта, успешно взаимодействующий с существующей инфраструктурой COM. Однако есть еще вопросы, которые должны быть решены в рамках COM. Это, например, языково-независимое описание и публикация интерфейсов, взаимодействие кода клиента и кода объекта, исполняемых в разных процессах, на разных машинах и другие.

Типы серверов

Существует три типа серверов COM.

1. Сервер в процессе (*in-process* server, называемый также *in-proc* сервером). Объекты этого сервера исполняются в процессе клиента и реализуются внутри DLL или OCX, загружаемой на машине клиента. В реестре путь к *in-proc* серверу записывается под ключом InProcServer32.

2. Сервер вне процесса (*out-of-process*), называемый также локальным сервером (*local server*). Объекты этого сервера исполняются в отдельном (другом) процессе на машине клиента. Объекты сервера реализуются внутри файла типа EXE. В реестре локальный сервер идентифицируется ключом LocalServer32.

3. Удаленный сервер (*remote server*). Объекты этого сервера исполняются в процессе на другой машине, поэтому эти серверы являются серверами типа *out-of-process*. Объекты удаленного сервера реализуются внутри файла типа DLL или EXE. В реестре эти серверы не записываются — они являются серверами *in-process* или локальными серверами на другой машине, и записываются в реестре ее операционной системы.

Маршалинг

В основе COM лежит использование COM-объектов через указатель на интерфейс. Когда объект реализован внутри *in-proc* сервера, код клиента и объекта расположены в одном виртуальном адресном пространстве, и указатель на интерфейс является действительным адресом этого пространства. Что происходит, когда объект реализуется в другом адресном пространстве (для локальных и удаленных серверов)? Адрес в другом адресном пространстве нельзя использовать в данном адресном пространстве — он просто не имеет никакого смысла.

Для реализации указателя на интерфейс в другом адресном пространстве (в другом процессе) используется так называемый маршалинг (*marshaling*).

Примечание: вообще говоря, маршалинг выполняется не только для объекта в другом адресном пространстве, но в некоторых случаях для объекта в другом потоке, поэтому под маршалингом лучше понимать реализацию указателя, пересекающего границы текущего потока.

Общая схема реализации указателя на интерфейс объекта, расположенного в другом процессе, приведена на рисунке 14.

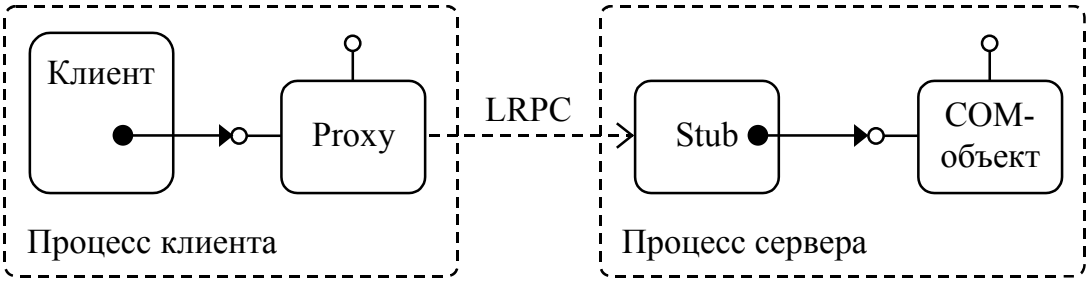


Рисунок 14 - Маршалинг интерфейса

Как видно из рисунка, между клиентом и COM-объектом находятся два вспомогательных элемента — заместитель (proxy) и заглушка (stub).

Так как клиентский указатель на интерфейс не может прямо указывать на интерфейс объекта, он указывает на заместитель объекта внутри клиентского процесса. Заместитель является обычным COM-объектом, реализующим те же интерфейсы, что и COM-объект, находящийся в другом процессе.

Однако заместитель не реализует интерфейсы объекта, он лишь представляет их. Вызов клиентом метода интерфейса на самом деле вызывает исполнение кода заместителя, называемого кодом маршалинга или просто маршалером (*marshaler*). Заместитель принимает переданные клиентом параметры метода, упаковывает их в некоторую стандартную форму, а затем при помощи LRPC передает запрос и его параметры заглушке, находящейся в другом процессе.

Заглушка, получив запрос, распаковывает параметры и вызывает необходимый метод COM-объекта. Получив ответ, заглушка упаковывает его, и передает заместителю, который, в свою очередь, возвращает ответ клиенту, иначе говоря, выполняет демаршалинг (*unmarshaling*).

Некоторый промежуточный формат данных, используемый для пересылки, необходим в случае, когда клиент и объект располагаются на разных машинах, которые, в свою очередь, могут работать в разном операционном окружении (иметь разную платформу) и иметь разное представление типов данных. Маршалинг, таким образом, выполняет синхронизацию типов разных платформ наиболее подходящим образом. Кстати, стандартный формат типов данных в точности соответствует типам языка *Visual Basic 6.0*, а если точнее, — типам *автоматизации*.

В среде *Microsoft Visual Studio* код маршалинга и демаршалинга генерируется автоматически. При необходимости, например, для увеличения производительности, может быть использован специализированный маршалинг (*custom marshaling*), при этом разработчик должен предоста

вить собственную реализацию интерфейса IMarshal. По некоторым сведениям, специализированный маршалинг используется приложениями *Microsoft Office*.

Апартаменты

Процесс состоит из виртуального адресного пространства, кода, данных и системных ресурсов. Код процесса исполняется посредством потоков. Изначально каждый процесс имеет первичный поток, однако процессы могут создавать другие потоки, которые могут исполнять как разные последовательности кодов, так и одни и те же.

Потоковая модель COM предполагает, что все COM-объекты процесса поделены на группы, называемые апартаментами. COM-объект существует ровно в одних апартаментах. Это означает, что прямой вызов метода объекта возможен только из потока, который принадлежит данным апартаментам, в противном случае используется *маршалинг*.

Существует два типа апартаментов.

1. Однопоточные апартаменты (STA, *single-threaded apartments*). Они состоят из одного потока. Методы объектов этих апартаментов вызываются прямо из этого потока в порядке очереди, которая управляется менеджером очередей (message queue).

2. Многопоточные апартаменты (MTA, *multithreaded apartments*). Они состоят из одного или более потоков. Методы объектов многопоточных апартаментов вызываются прямо из любого потока этих апартаментов. Вызовы методов синхронизируются самими объектами.

Апартаменты являются важной (и сложной) частью COM. Они упрощают написание безопасного параллельно выполняющегося кода без явной реализации в нем синхронизации.

IDL

Для описания интерфейсов используется язык IDL (*Interface Definition Language*), произошедший, в свою очередь от языка ODL (*Object Definition Language*). Эти языки имеют синтаксис, схожий с синтаксисом языка Си. Файлы, описывающие интерфейсы, имеют расширение .idl, и компилируются при помощи, например, MIDL (Microsoft IDL, расширение языка IDL).

Каждый объект IDL описывается при помощи двух конструкций. Первая конструкция заключается в квадратные скобки и представляет собой *метаданные*. Вторая конструкция описывает собственно объект в стиле языка Си. В качестве примера рассмотрим описание интерфейса IBeeper на языке IDL. Сначала описываются метаданные:

```
[
    object,
    uuid(8C530821-9A3E-474F-B548-B2C878A62358),
    helpstring("IBeeper Interface"),
    pointer_default(unique)
]
```

Метаданные, как видно, содержат GUID интерфейса, краткое описание (*helpstring*) и некоторую дополнительную информацию.

Непосредственно за этим блоком должно следовать описание объекта, в нашем случае интерфейса:

```
interface IBeeper : IUnknown
{
    [helpstring("method Beep")] HRESULT Beep();
    [propget, helpstring("property Tone")]
    HRESULT Tone([out, retval] short *pVal);
    [propput, helpstring("property Tone")]
    HRESULT Tone([in] short newVal);
};
```

Как видим, каждому элементу интерфейса также предшествуют метаданные, заключенные в квадратные скобки. Они включают в себя краткое описание, указание на тип элемента (propget — чтение свойства, propput — запись свойства), а также другую информацию. Кроме того, в описании параметров функций интерфейса также встречается метаинформация:

`[in]` — параметр является входным (параметр по значению);

`[out]` — параметр является выходным (параметр по ссылке);

`[out, retval]` — параметр является возвращаемым значением.

Описание кокласса Beeper на языке IDL может иметь примерно следующий вид:

```
[
    uuid(DAC2536A-C7BB-4CB4-BD9F-0AC1D2ADDE38),
    helpstring("Beeper Class")
]
coclass Beeper
{
    [default] interface IBeeper;
};
```

Заметим, что с помощью языка IDL можно описывать также и другие объекты, например, константы, перечисления и синонимы типов.

Библиотеки типов

Информация о типе включает в себя описание всего, что нужно знать клиенту для обращения к сервисам COM-объекта. Чтобы предоставить информацию о типе клиентам, разработчик может и должен создавать и распространять библиотеки типов.

Библиотека типов — двоичный файл, содержащий стандартное описание интерфейсов, коклассов, констант, перечислений и т.п. Файл библиотеки типов имеет расширение .tlb (*type library*) или .olb (*object library*). Эта библиотека генерируется из файла .idl при помощи MIDL или приложения MkTypLib.exe. Библиотека типов может также быть составной частью COM-сервера (что характерно для серверов, создаваемых MSVB).

Чтобы получить доступ к библиотеке типов, сначала нужно получить указатель на интерфейс ITypeLib при помощи системной функции *LoadRegTypeLib*, передав ей GUID библиотеки, версию и информацию о локали. Далее при помощи методов интерфейса можно получить отдельные элементы информации.

Так, при помощи метода *GetTypeInfoOfGUID* можно получить указатель на интерфейс ITypeInfo, с помощью которого далее можно получить описание методов и атрибутов, используя методы *GetFuncDesc* или *GetTypeAttr*.

Заметим, что информация о типе чаще всего нужна клиентам, использующим интерфейс IDispatch.

В *Microsoft Visual Studio 6.0* есть также приложение OleView, помощью которого можно посмотреть содержимое библиотеки типов.

При создании сервера COM при помощи MSVS файл .idl создается и компилируется автоматически с получением двоичного файла библиотеки типов, а также кода маршалинга в виде отдельной DLL или в составе сервера. При создании сервера COM при помощи MSVB библиотека типов является частью сервера.

Повторное использование объектов

Повторное использование объектов COM (*object reusing*) — это возможность использовать одни COM-объекты в составе других. Различают два вида повторного использования.

1. Включение/делегирование (*Containment*/*Delegation*). В этом случае внешний объект выступает как клиент внутреннего объекта, а внутренний объект делегирует свои методы внешнему объекту (рисунок 15).

Клиент вызывает методы внешнего объекта, а тот, в свою очередь, вызывает соответствующие методы внутреннего объекта.

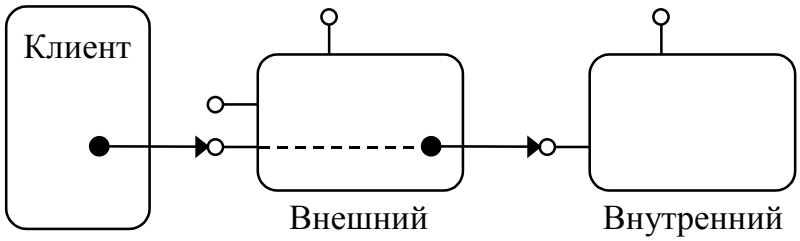


Рисунок 15 - Делегирование интерфейса внутреннего COM-объекта

Заметим, что обозначение объектов как *внешних* и *внутренних* является *условным*, так как фактически все COM-объекты являются самостоятельными единицами и не могут быть включены один в другой.

Делегирование особенно характерно при программировании с помощью MSVB, и это единственный способ повторного использования для этой системы программирования.

2. Агрегация (*Aggregation*). Включение замедляет процесс вызова метода внутреннего объекта. При агрегировании внешний объект выставляет интерфейсы внутреннего объекта как свои собственные (рисунок 16). Клиент, обращаясь к интерфейсу внешнего объекта, на самом деле получает указатель на интерфейс внутреннего объекта.

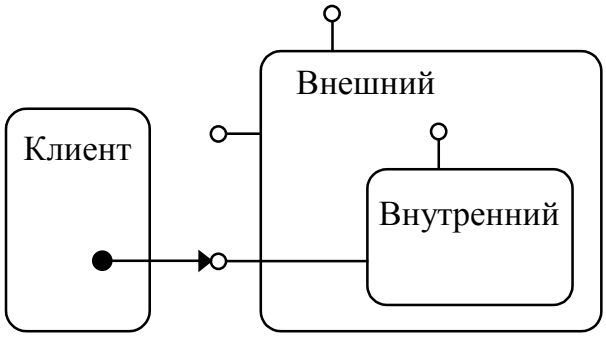


Рисунок 16 - Агрегация внутреннего COM-объекта

Агрегирование требует специальной организации внешнего объекта, которая в рамках данного пособия не рассматривается. Заметим только, что в некоторых функциях COM требуется указатель на интерфейс IUnknown агрегирующего (внешнего) объекта. Это дает возможность внешнему объекту исследовать интерфейсы агрегируемого (внутреннего) объекта и экспортировать их.

Использование ATL 3.0

Для создания серверов COM можно использовать как MSVS, так и MSVB. В данном разделе описывается создание сервера COM средствами MSVS с использованием ATL.

ATL (*ActiveX Template Library*) — это набор шаблонов для создания COM-объектов, являющийся частью MSVS. Для создания сервера COM на основе шаблонов используется мастер *ATL Project Wizard*.

Практическая работа №4

Для изучения ATL MSVS будем создавать сервер DLL для нашего простейшего объекта Beeper.

Создание сервера

Открываем MSVS, выбираем «*create Project*», язык «*Visual C++*», раздел «*ATL*», шаблон «*ATL Project*». Имя проекта BEEPC. Папка проекта предположительно C:\BEEPC. Нажимаем кнопки OK и Next. Заметим, что название проекта становится именем сервера, которое будет регистрироваться в реестре, поэтому к выбору названия проекта нужно подходить обдуманно.

На рисунке 17 приведен вид мастера *ATL Project Wizard*.

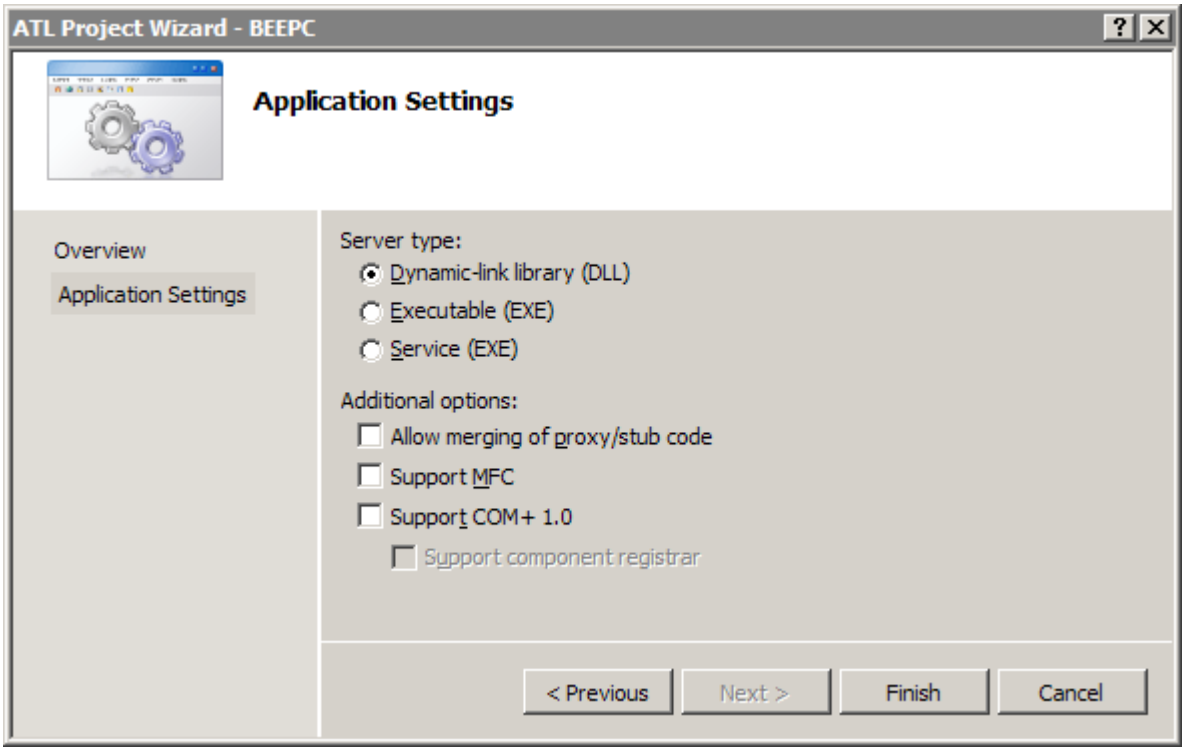


Рисунок 17 - Мастер ATL Project Wizard

В первую очередь нужно выбрать тип сервера — DLL или EXE. Заметим, что EXE сервер, в свою очередь, может быть приложением или чистым сервером (сервисом). Мы выбираем DLL (по умолчанию).

Флажок «*Allow merging of proxy/stub code*» позволяет включить код маршалинга непосредственно в проект. Флажок «*Support MFC*» управляет поддержкой MFC (*Microsoft Foundation Classes*). Мы оставляем флажки выключенными. Нажимаем кнопку Finish.

По завершении работы мастера создается проект, состоящий из множества файлов. Рассмотрим назначение только некоторых файлов.

Важнейшим файлом проекта является файл BEEPC.idl:

```
import "oaidl.idl";
import "ocidl.idl";
[
    uuid(A8568AD9-7027-4F36-9DDE-3CC1FEA1D36E),
    version(1.0),
    helpstring("BEEPC 1.0 Type Library")
]
library BEEPCLib
{
    importlib("stdole2.tlb");
};
```

Файл .idl описывает библиотеку типов, которая создается одновременно с созданием сервера. При построении проекта этот файл компилируется при помощи MIDL с получением файла библиотеки типов (в нашем случае BEEPCLib.tlb) и кода маршалинга.

Как видно из приведенного кода, сейчас файл BEEPC.idl содержит описание только одного объекта — собственно библиотеки типов. Директивы *import* и *importlib* описывают включение в проект необходимых структур данных и интерфейсов инфраструктуры COM.

Второй файл, который мы рассмотрим, — основной файл проекта BEEPC.cpp. Он содержит реализацию функций DLL. Заметим, что библиотека ATL берет на себя всю рутинную работу по созданию сервера. Вот пример реализации функция *DllGetClassObject*:

```
// Returns a class factory to create an object of the requested type
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    return _AtlModule.DllGetClassObject(rclsid, riid, ppv);
}
```

Кроме того, проект содержит также проект DLL BEEPCPS, который генерирует код маршалинга (прокси и заглушки).

Создание COM-объекта

После создания сервера нужно добавить в него необходимое количество COM-объектов. Выбираем в меню *Project* - *Add Class*. Появится диалог *Add Class* (рисунок 18).

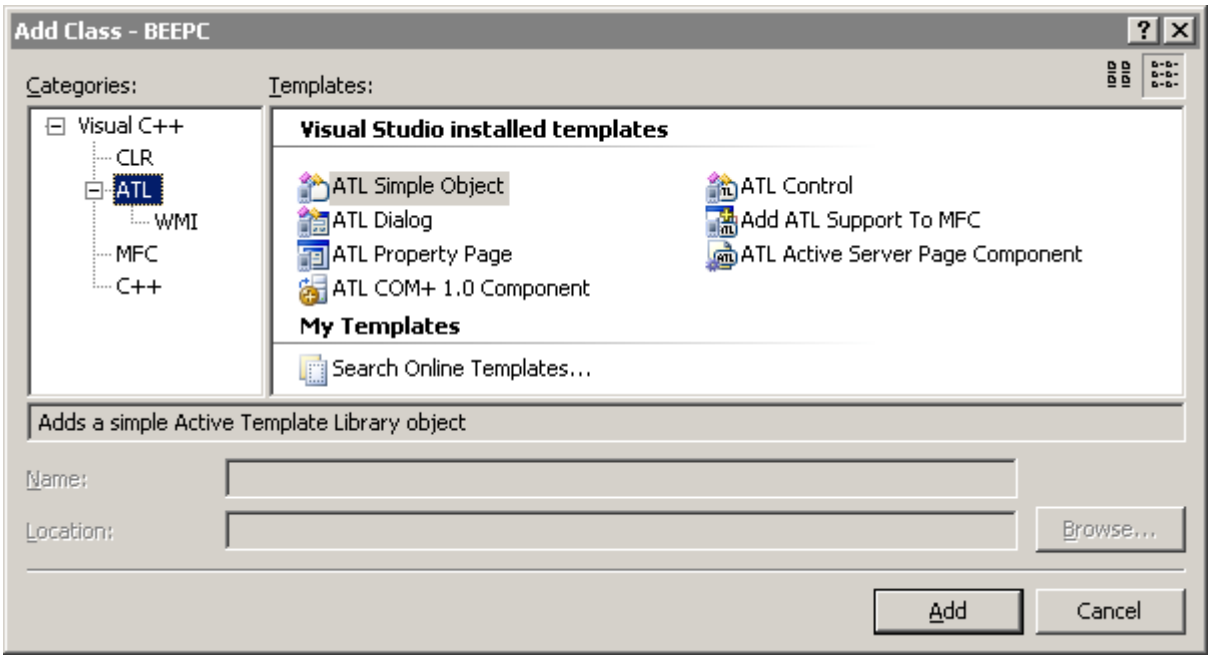


Рисунок 18 - Выбор шаблона COM-объекта

Здесь можно выбрать тип создаваемого объекта COM. Мы выбираем *ATL Simple Object* и нажимаем кнопку Add.

Появляется мастер создания COM-объекта (рисунок 19).

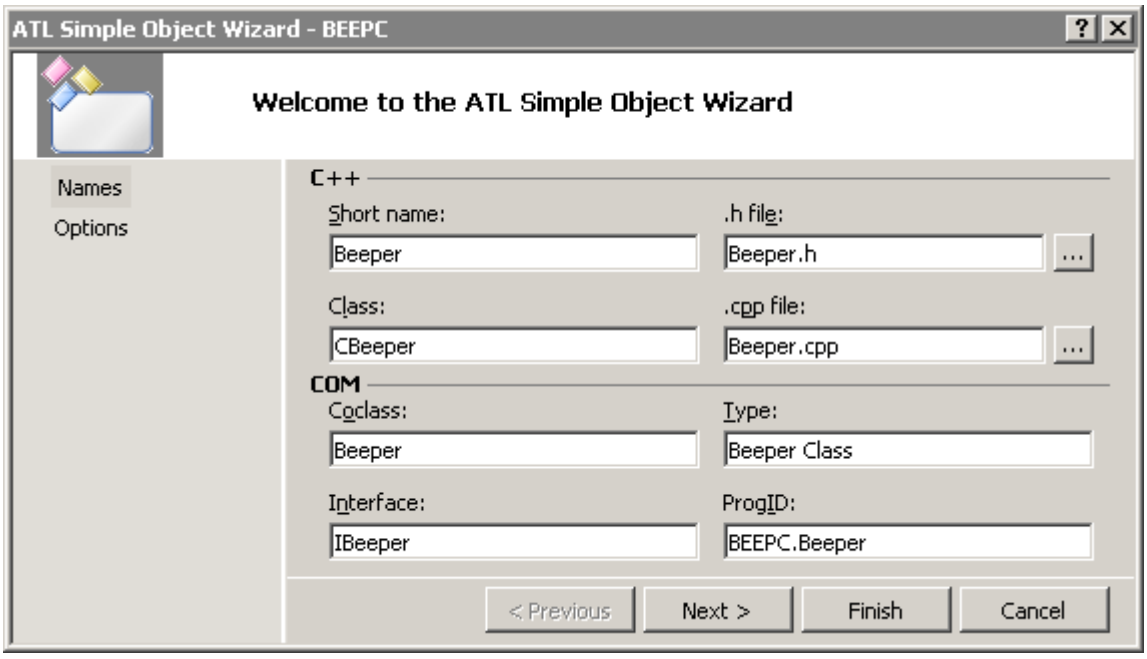


Рисунок 19 - Определение имен COM-объекта

На первом шаге определяются имена классов и интерфейсов. Вписываем название Beeper в поле *Short Name*. Нажимаем кнопку Next.

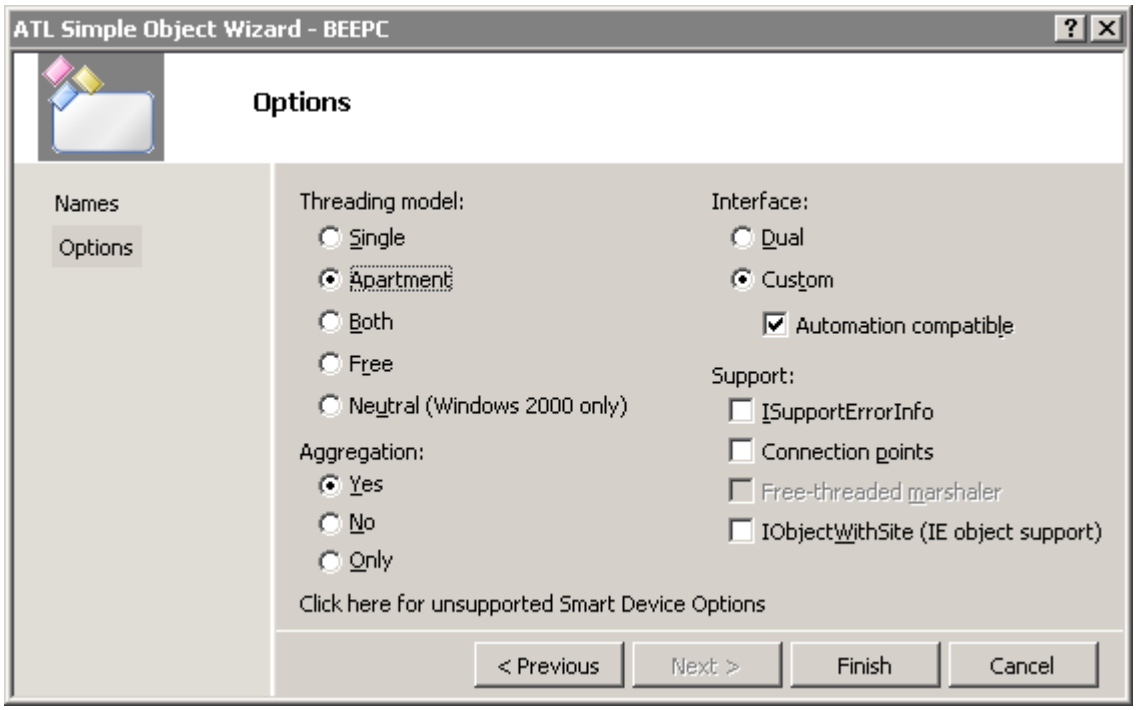


Рисунок 20 - Свойства кокласса

На следующем шаге выбирается потоковая модель (*Threading model*), базовый интерфейс (*Interface*) и агрегация (*Aggregation*) объекта.

В потоковой модели первые две опции соответствуют однопоточным апартаментам, а две вторые — многопоточным. Мы выбираем значение по умолчанию *Apartment*.

Интерфейс объекта может быть производным либо от интерфейса IDispatch, либо от интерфейса IUnknown. В первом случае получается так называемый дуальный интерфейс (*Dual*), а во втором случае — пользовательский (*Custom*). Мы выбираем *Custom* (по умолчанию *Dual*), и устанавливаем флажок *Automation compatible* (совместимый с автоматизацией).

Агрегации объектов у нас не будет, тем не менее, оставляем эту возможность (используем значение по умолчанию Yes).

Флажок «*ISupportErrorInfo*» создает объект, наследующий соответствующий интерфейс ATL, отвечающий за поддержку расширенной информации об ошибке.

Флажок «*Connection Points*» создает объект, наследующий интерфейсы ATL, отвечающие за создание точек соединения (объектов с подключениями), то есть за события объекта.

Нажмем кнопку Finish. В результате в проекте появятся два новых файла — Beeper.h (описание кокласса CBeeper) и Beeper.cpp (реализация

методов кокласса). Кроме того, в библиотеку типов добавляется описание интерфейса и кокласса, вносятся также и другие изменения.

Рассмотрим сначала библиотеку типов BEEPC.idl. Описание интерфейса обычно добавляется вне библиотеки типов, а описание кокласса всегда располагается внутри.

```
import "oaidl.idl";
import "ocidl.idl";
[
    object, uuid(D7B63DBA-7C99-4A75-9BBD-79E11136414E),
    oleautomation, nonextensible,
    helpstring("IBeeper Interface"),
    pointer_default(unique)
]
interface IBeeper : IUnknown {
};
[
    uuid(A8568AD9-7027-4F36-9DDE-3CC1FEA1D36E),
    version(1.0),
    helpstring("BEEPC 1.0 Type Library")
]
library BEEPCLib
{
    importlib("stdole2.tlb");
    [
        uuid(BF120BF3-3A8E-4562-8ACE-3D4A05C2DC04),
        helpstring("Beeper Class")
    ]
    coclass Beeper
    {
        [default] interface IBeeper;
    };
};
```

Рассмотрим теперь описание кокласса (файл Beeper.h).

```
class ATL_NO_VTABLE CBeeper :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CBeeper, &CLSID_Beeper>,
    public IBeeper
{
```

Кокласс наследует три элемента. Первые два являются шаблонами ATL и они скрывают от разработчика детали реализации стандартного базового интерфейса IUnknown.

Рассмотрение классов ATL выходит за рамки данного пособия.

Далее описывается конструктор и следующая конструкция:

```
DECLARE_REGISTRY_RESOURCEID(IDR_BEEPER)
```

Это макроопределение определяет идентификатор ресурса, в котором хранится описание класса.

Далее следует макроопределение

```
BEGIN_COM_MAP(CBeeper)
    COM_INTERFACE_ENTRY(IBeeper)
END_COM_MAP()
```

которое описывает экспортируемые интерфейсы. Каждый такой интерфейс описывается макроопределением COM_INTERFACE_ENTRY. Клиент может получить указатель только на тот интерфейс, который есть в данной таблице.

Следующее макроопределение

```
DECLARE_PROTECT_FINAL_CONSTRUCT()
```

предназначено для защиты объекта от уничтожения, если во время выполнения функции FinalConstruct будет неудачно создан агрегированный объект.

Непосредственно за этим макроопределением располагаются функции *FinalConstruct* и *FinalRelease*, используемые при агрегировании:

```
    HRESULT FinalConstruct()
    {
        return S_OK;
    }
    void FinalRelease()
    {
    }
```

Наконец, далее следует собственно сам кокласс, который сейчас ничего не содержит:

```
public:
};
```

В файле Beeper.cpp мы ничего не увидим, поскольку интерфейс и, соответственно, кокласс, не описывают никаких методов.

Константа CLSID_Beeper определена в файле BEEP_i.c.

Кроме этого, после компиляции проекта создается файл Beeper.rgs, который содержит описание информации, которая должна быть внесена в реестр для регистрации.

Добавление метода

Для добавления методов и свойств в интерфейс COM-объекта в ATL используются различные мастера. Заметим, что добавить метод или свойство можно также вручную. Удалить метод или свойство можно только вручную, а для этого нужно знать, что и куда добавляется при создании метода или свойства.

Сначала из вкладки *Solution Explorer* нужно перейти на вкладку *ClassView* (рисунок 21, внизу).

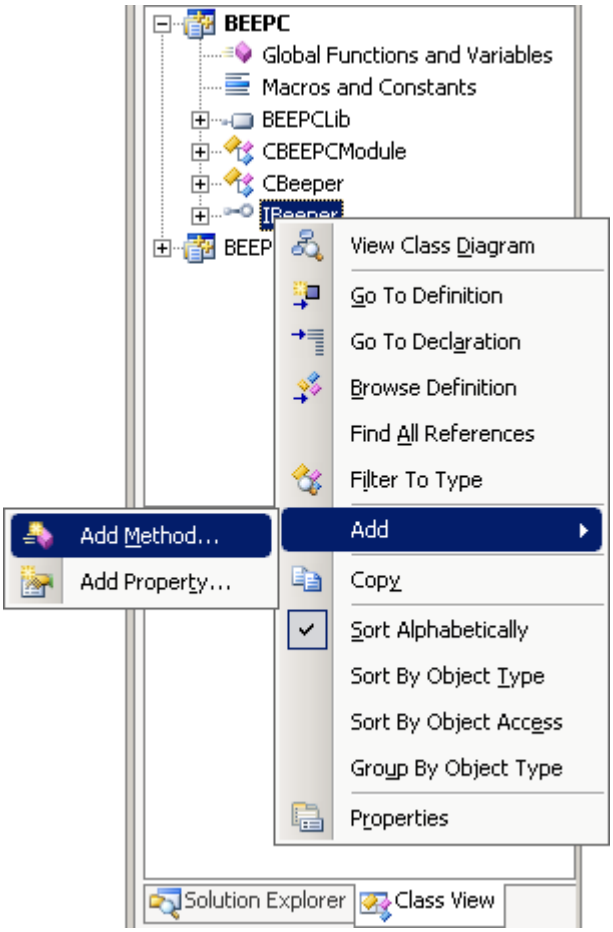


Рисунок 21 - Выбор добавления в интерфейс метода или свойства

Далее нужно найти интерфейс IBeeper и щелкнуть на него правой кнопкой мыши (вызвать контекстное меню).

Может оказаться, что вы увидите два интерфейса IBeeper. Выбирать необходимо тот, который обозначает именно интерфейс — он отмечен значком в виде ручки, такой же, какой обозначают интерфейс на графическом обозначении COM-объекта (рисунок 21, выделенный интерфейс IBeeper).

Выбираем в контекстном меню *Add* - *Add Method*. Появится мастер добавления метода *Add Method Wizard* (рисунок 22).

Рисунок 22 - Добавление метода в интерфейс

В поле Method Name введем название метода Beep. К методу можно добавить необходимое число параметров, если выбрать тип в списке *Parameter type* и вписать название параметра в поле *Parameter name*. Направление параметра выбирается при помощи флажков *in*, *out* и *retval*. Кнопка Add добавляет параметр, а кнопка Remove — удаляет.

Если нажать кнопку Next, появится второй шаг мастера, в котором можно выбрать дополнительные атрибуты (рисунок 23).

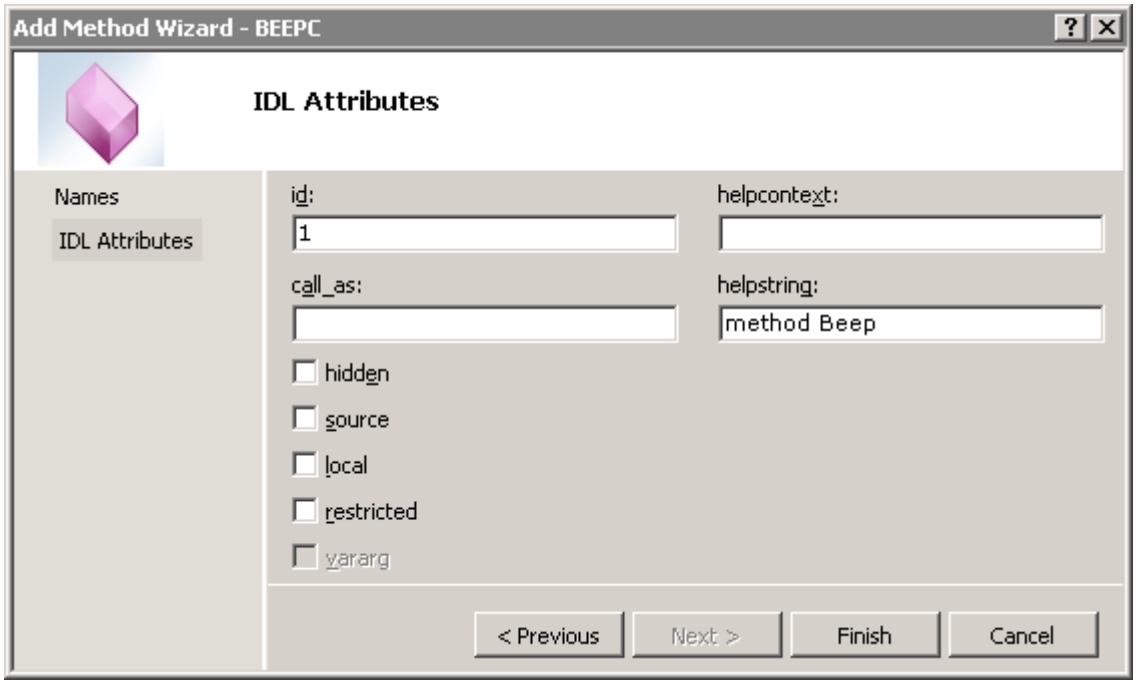


Рисунок 23 — Атрибуты метода

Нажимаем кнопку Finish.

В результате произошли изменения в трех файлах. Во-первых, в описании интерфейса в файле BEEPC.idl появилось описание метода:

```
interface IBeeper : IUnknown {
    [id(1), helpstring("method Beep")] HRESULT Beep(void);
};
```

Во-вторых, описание метода появилось также в файле Beeper.h:

```
public:
    STDMETHOD(Beep)(void);
};
```

В-третьих, в файле Beeper.cpp появилась реализация метода:

```
STDMETHODIMP CBeeper::Beep()
{
    // TODO: Add your implementation code here
    return S_OK;
}
```

Добавление свойства

Добавление свойства выполняется аналогично добавлению метода.

Открываем контекстное меню интерфейса IBeeper (рисунок 21) и выбираем в нем *Add* - *Add Property*. Появится диалог для добавления свойства (рисунок 24).

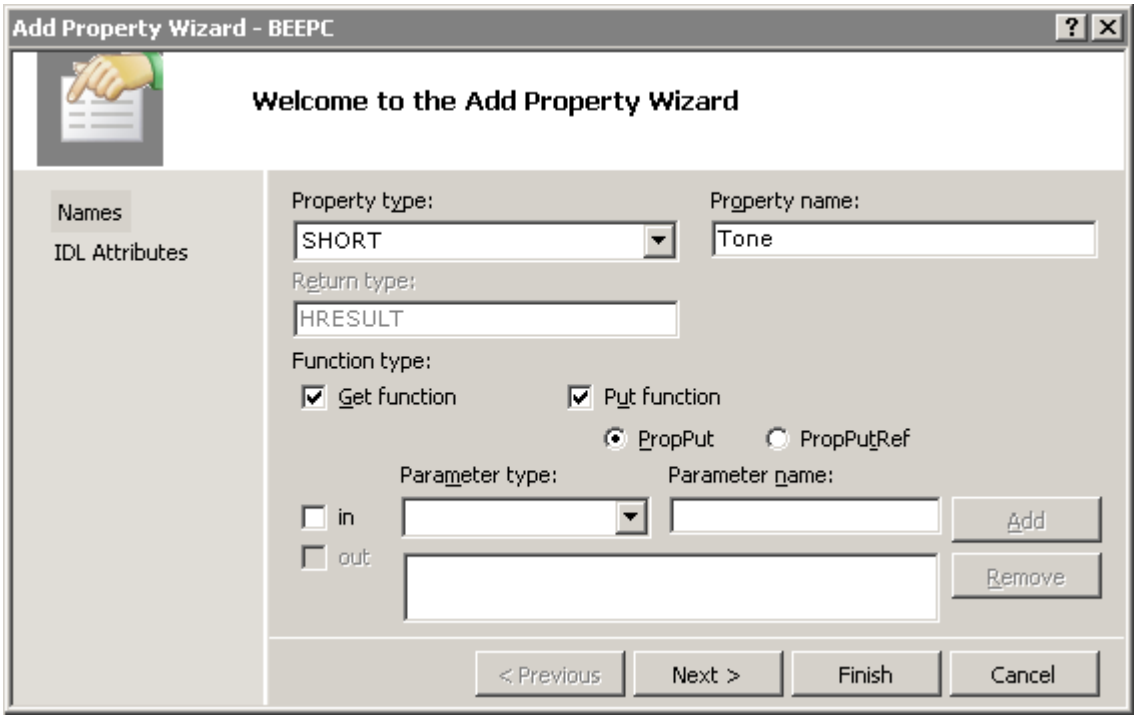


Рисунок 24 - Добавление в интерфейс свойства

В этом диалоге нужно выбрать тип свойства (список *Property type*), ввести название свойства (поле *Property name*), а при необходимости также добавить параметры свойства.

При помощи флажков «*Get Function*» и «*Put Function*» можно выбрать, какие функции будут созданы для свойства. Если первый флажок выбран, будет создана функция для чтения свойства. Если второй флажок выбран, будет создана функция для записи свойства. Таким образом, можно создать свойство для чтения и записи, только для чтения и только для записи.

На втором шаге мастера (кнопка Next) задают атрибуты свойства, однако можно это сделать и позже, так же, как и для метода.

Нажимаем кнопку Finish.

Изменения также внесены в три файла.

В файле BEEPC.idl появляется описание функций свойства (здесь описание функций разбито на две строки):

```
interface IBeeper : IUnknown {
    [id(1), helpstring("method Beep")] HRESULT Beep(void);
    [propget, helpstring("property Tone")]
        HRESULT Tone([out, retval] SHORT* pVal);
    [propput, helpstring("property Tone")]
        HRESULT Tone([in] SHORT newVal);
};
```

В файле Beeper.h также появились описания функций свойства:

```
public:
    STDMETHOD(get_Tone)(/*[out, retval]*/ short *pVal);
    STDMETHOD(put_Tone)(/*[in]*/ short newVal);
    STDMETHOD(Beep)();
};
```

Наконец, в файле Beeper.cpp появились функции свойства. Здесь они не приводятся, так как не содержат ничего нового.

Осталось наполнить содержанием функции кокласса. Переходим в модуль Beeper.h, добавляем секцию private в конце описания кокласса и описываем элемент данных для хранения частоты:

```
public:
    . . .
    STDMETHOD(Beep)();
private:
    // Частота
    short m_tone;
};
```

В конструкторе кокласса устанавливаем начальное значение:

```
CBeeper()
{
    m_tone = 0;
}
```

Переходим в модуль Beeper.cpp и описываем функции кокласса:

```
STDMETHODIMP CBeeper::Beep()
{
    WCHAR buff[20];
    wsprintfW( buff, L"Beep %d", m_tone );
    MessageBoxW( 0, buff, L"Beeper", MB_OK );
    return S_OK;
}

STDMETHODIMP CBeeper::get_Tone(short *pVal)
{
    *pVal = m_tone;
    return S_OK;
}

STDMETHODIMP CBeeper::put_Tone(short newVal)
{
    m_tone = newVal;
    return S_OK;
}
```

Компилируем проект с получением сервера BEEPC.dll.

Тестирование сервера

Для тестирования сервера добавим проект на языке Visual Basic.

Выбираем в меню *File* - *Add* - *New Project*. Выбираем язык *Visual Basic*, раздел *Windows*, шаблон *Console Application*.

Вводим название проекта TestBC и нажимаем кнопку OK.

В окне *Solution Explorer* щелкаем правой кнопкой мыши на название проекта TestBC, выбираем *Set as StartUp Project* (делаем проект стартовым).

Теперь нам нужно добавить ссылку (*reference*) проекта TestBC на проект BEEPC. На вкладке *Solution Explorer* должен быть выбран проект TestBC.

Выбираем в меню *Project* - *Add Reference*.

Появится диалог для выбора ссылок на различные виды объектов (рисунок 25).

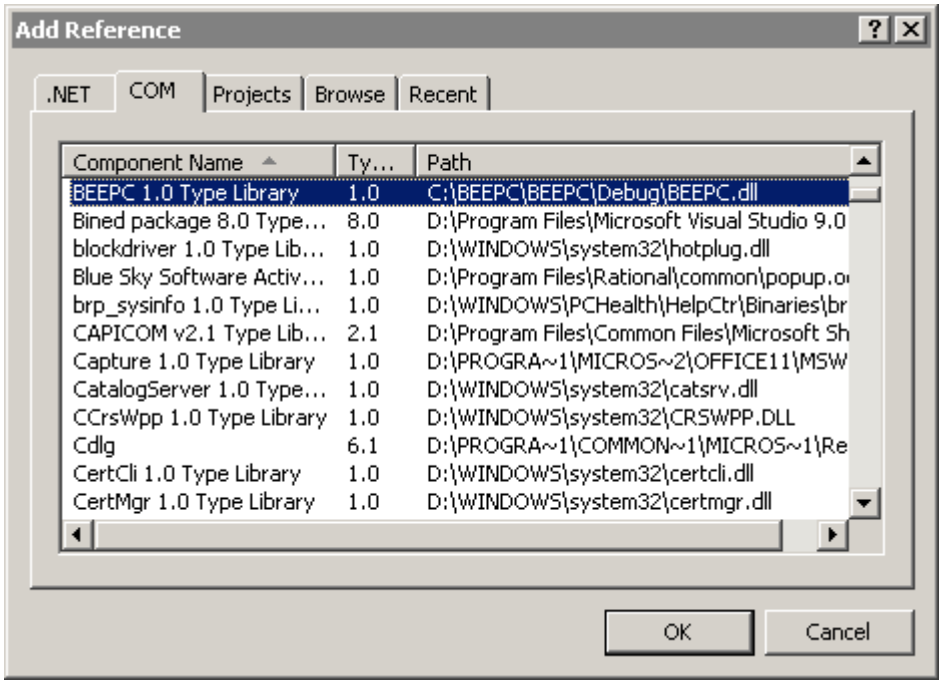


Рисунок 25 - Диалог References

В этом диалоге нужно выбрать вкладку COM и найти в списке строку с записью нашего сервера, после чего нажать кнопку OK.

Модуль проекта Module1.vbp содержит процедуру *Main*, которая является стартовым объектом проекта. Содержание модуля должно быть следующим:

```
Module Module1
    Sub Main()
        ' Объявляем ссылку на объект
        Dim B As BEEPCLib.Beeper
        ' Создаем объект
        B = New BEEPCLib.Beeper
        ' Вызываем свойство
        B.Tone = 80
        ' Вызываем метод
        B.Beep()
    End Sub
End Module
```

После запуска этого проекта на выполнение можно убедиться в том, что COM-объект сервера создается успешно, и мы можем обращаться к его свойствам и методам.

Заметим, что всю рутинную работу по созданию COM-объекта система программирования *Visual Basic* взяла на себя (в операторе *New*).

COM+

Развитие концепций объектно-ориентированного программирования фирмой Microsoft в конечном итоге привело к созданию клиент-серверной технологии, называемой .NET (DOT NET). На этом длинном пути Microsoft разработала множество архитектур, средств разработки и продуктов, направленных на упрощение процесса программирования конкретным программистом. Если изначально технологии Microsoft были ориентированы на рабочее место, то постепенно фирма сконцентрировала свои усилия на уровень группы и далее на уровень распределенных приложениях. Инфраструктура COM явилась важным шагом, предопределившим концепцию программных компонентов, независимо работающих в гетерогенных распределенных средах, таких, какие объединяет сеть Интернет.

Рассматривая конкретные продукты от Microsoft, следует отметить *Microsoft Transaction Server*, *Microsoft Message Queue*, *Distributed Network Architecture* (DNA) и другие.

Microsoft Transaction Server (MTS, сервер транзакций) упрощает создание серверных приложений с помощью COM. Компонент COM импортируется в пакет MTS, обеспечивающий необходимые свойства. При этом MTS берет на себя решение таких сложных системных задач, как масштабируемость, безопасность, многопоточность, обеспечение транзакций.

Microsoft Message Queue (MSMQ, очередь сообщений) представляет собой простую модель построения распределенных систем. Приложение создает сообщение и отправляет его в очередь. Другое приложение может считать сообщение из очереди и послать другое сообщение и т.д.

Очередь сообщений использует RPC (*remote procedure call*) и технологию DCOM для высокоуровневого обмена информацией между приложениями.

COM+ следует рассматривать как следующее поколение компонентной архитектуры. COM+ интегрирует MTS и COM и обеспечивает альтернативу вызовам COM, используя механизм сообщений, основанных на MSMQ. В результате получилась цельная система, в которой упрощается создание как серверных, так и клиентских приложений.

Если COM является *инфраструктурой*, то COM+ — это *промежуточное программное обеспечение* (*middleware*). Важнейшая особенность COM+ заключается в архитектуре, называемой *перехватом*. Она позволяет вмешиваться промежуточному ПО в работу приложений только в случае необходимости, а не постоянно. Другой особенностью COM+ является декларативное программирование, основанное на атрибутах.

Приложения COM+

В COM+ вводится понятие *приложения*. Этот термин чрезвычайно перегружен, однако в COM+ он имеет специальное значение. Приложение представляет собой группу классов, разделяющих некоторое определенное множество атрибутов безопасности, активизации, и т.п.

В COM существует понятие сервера, в качестве которого выступает DLL или EXE файл. В COM+ концепция сервера менее важна. Все классы в COM+ должны быть реализованы в DLL-сервере. Если вам требуется EXE-сервер, то COM+ будет запускать их в стандартном «суррогате» DLLHOST.exe.

В основе использования COM+ лежит также понятие *компонента*. Этот термин является еще более перегруженным, но для простоты мы будем понимать под ним бинарный (двоичный) модуль кода, который создает бинарный объект (согласно MSDN). Такое определение включает в себя COM-классы и DLL-сервер, однако мы будем полагать, что компонент однозначно соответствует коклассу. При этом компонент обязан отвечать следующим требованиям:

- реализация интерфейса IUnknown;
- реализация функций DLL, CLSID, IID, фабрики класса;
- саморегистрация;
- предоставление библиотеки типов;
- маршалинг интерфейсов (маршалинг дуальных интерфейсов посредством автоматизации).

Компонент, удовлетворяющий перечисленным требованиям, может участвовать в сервисах COM+, но сначала его нужно установить в приложение COM+. Это можно сделать с помощью административной консоли Component Services (Службы компонентов), либо программно с помощью соответствующего API.

После того, как компонент был установлен в приложение COM+, он становится сконфигурированным компонентом (*configured component*). Компонент, не установленный в приложение COM+, называется несконфигурированным компонентом (*unconfigured*).

Таким образом, формирование сервиса COM+ возможно только при наличии компонента, проще говоря — COM DLL.

Существует два типа приложений COM+. Главным является *приложение сервера*. Приложение сервера работает в своем собственном процессе стандартного суррогата DLLHOST.exe. Приложение сервера обеспечивает изоляцию ошибок и крах сервера не ведет к краху клиентов, так как они работают в разных адресных пространствах.

Библиотечное приложение работает в процессе клиента. Доступ к нему возможен только в пределах машины клиента.

Прокси-приложение содержит информацию, необходимую для доступа удаленному клиенту. Это файл, который может быть запущен на машине клиента для регистрации сервера.

Кроме того, имеется несколько предустановленных, системных приложений, которые не могут быть удалены.

Службы компонентов

Для управления приложениями COM+ в операционной системе имеется консоль Component Services (Службы компонентов), расположенная в панели управления (рисунок 26).

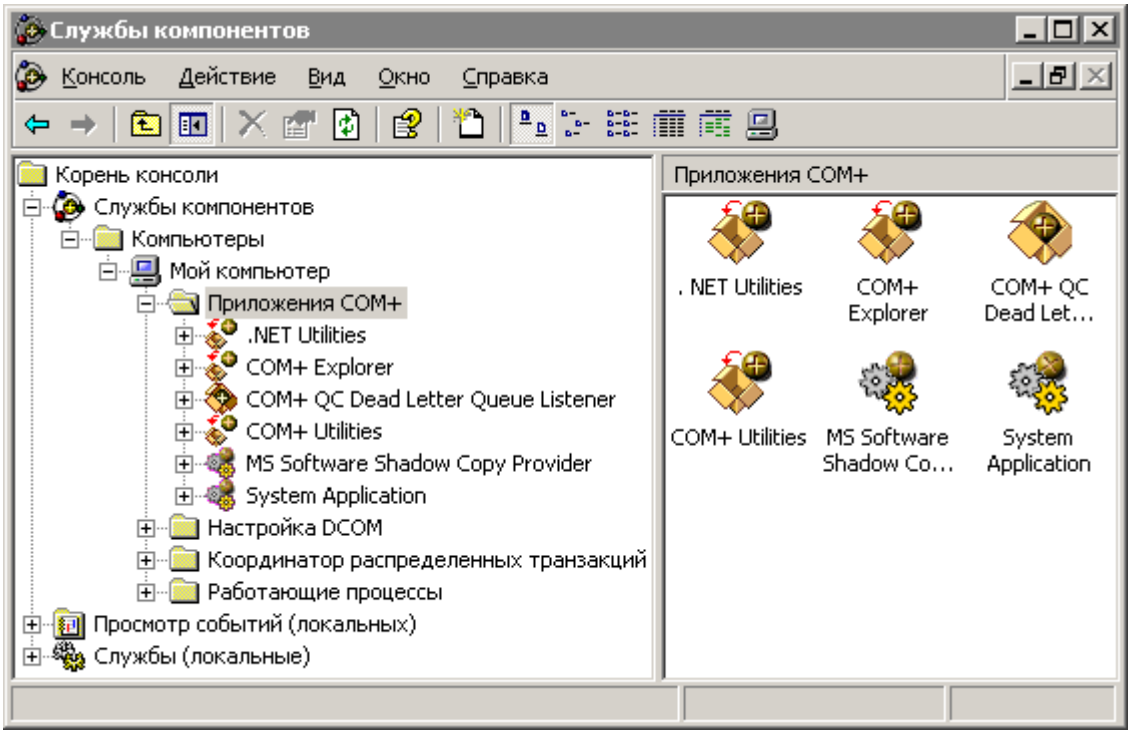


Рисунок 26 — Консоль «Службы компонентов»

Не следует путать сервисы COM+ с сервисами операционной системы. Сервисы COM+ — это обеспечиваемые ею свойства среды, в которой работают компоненты COM+. К этим свойствам относят безопасность, согласованность, транзакции, активизацию и другие.

Декларативное программирование на атрибутах

Программирование приложений COM+ отличается от того, чем обычно занимается программист. В COM+ не нужно писать программный код (если не учитывать написание компонентов). В COM+ используется декларативная модель, основанная на значениях атрибутов.

Компоненты COM+ работают внутри так называемого контекста.

Контекст нужно рассматривать как набор ограничений времени выполнения. Эти ограничения являются атрибутами и могут быть заданы в простейшем случае путем включения соответствующего флажка в диалоге, управляющем компонентом (рисунок 27).

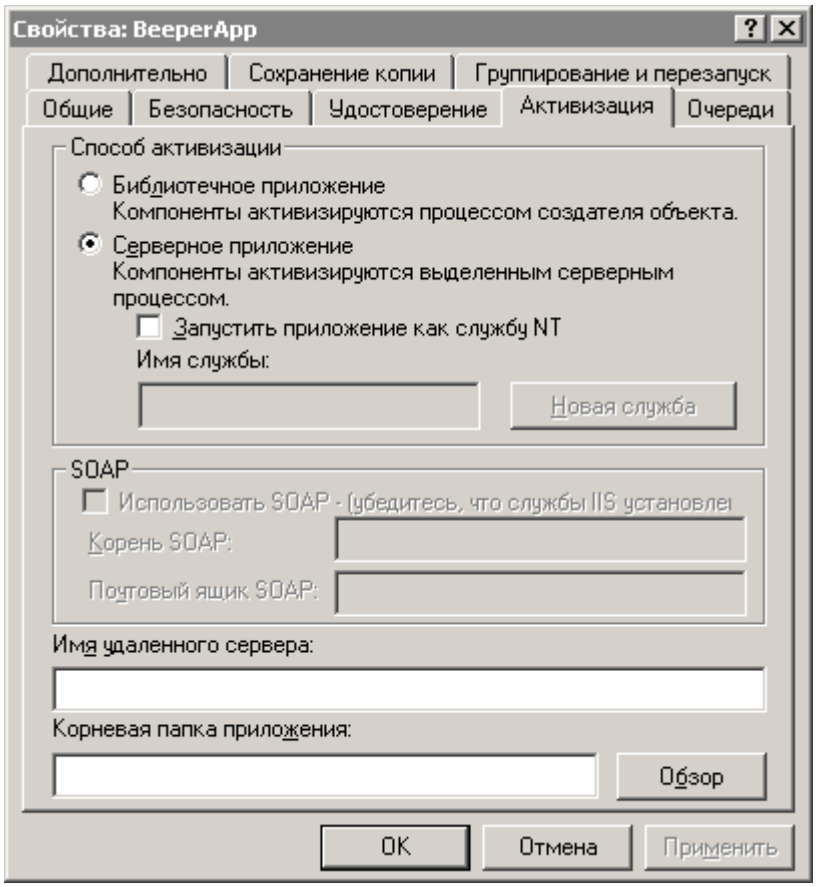


Рисунок 27 — Атрибуты компонента

Значения атрибутов хранятся в конфигурационной базе данных, называемой каталогом (рисунок 28).

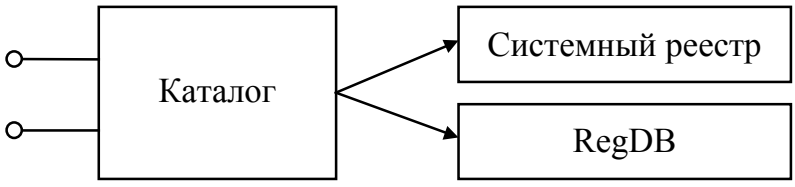


Рисунок 28 — Каталог COM+

Часть информации, относящаяся к COM, хранится, как и обычно, в системном реестре Windows. Конфигурационная информация приложений COM+ хранится в отдельной системной базе данных RegDB. Доступ к этой базе данных возможен через семейство системных объектов, известных под названием «менеджер каталогов» (*catalog manager*). Эти объекты доступны из программ и сценариев (скриптов).

Архитектура COM+

В основе COM+ лежат три основные концепции — *контекст*, *активизация* и *перехват*. Фундаментальным принципом COM+ является декларативное программирование, основанное на атрибутах. Программист с помощью атрибутов описывает среду, в которой должен работать компонент. Эта информация хранится в каталоге.

При создании объекта COM+ просматривает каталог и определяет, является ли компонент сконфигурированным. Если компонент записан в каталоге, он создается и размещается в контексте, соответствующем установленным атрибутам. Связывание объекта с его контекстом называется активизацией.

Если клиент компонента находится в контексте, отличном от контекста компонента, COM+ использует перехват, обеспечивая совместную работу несовместимых компонентов. Если клиент и компонент имеют одинаковые требования времени выполнения, компонент создается в контексте клиента и перехват не требуется.

Контекст представляет собой коллекцию объектов в пределах апартамента, имеющих одинаковые требования времени выполнения. Каждый COM-объект связывается ровно с одним контекстом, а контекст располагается в точности в одном апартаменте. Объекты в разных апартаментах обязательно имеют разный контекст (рисунок 29).

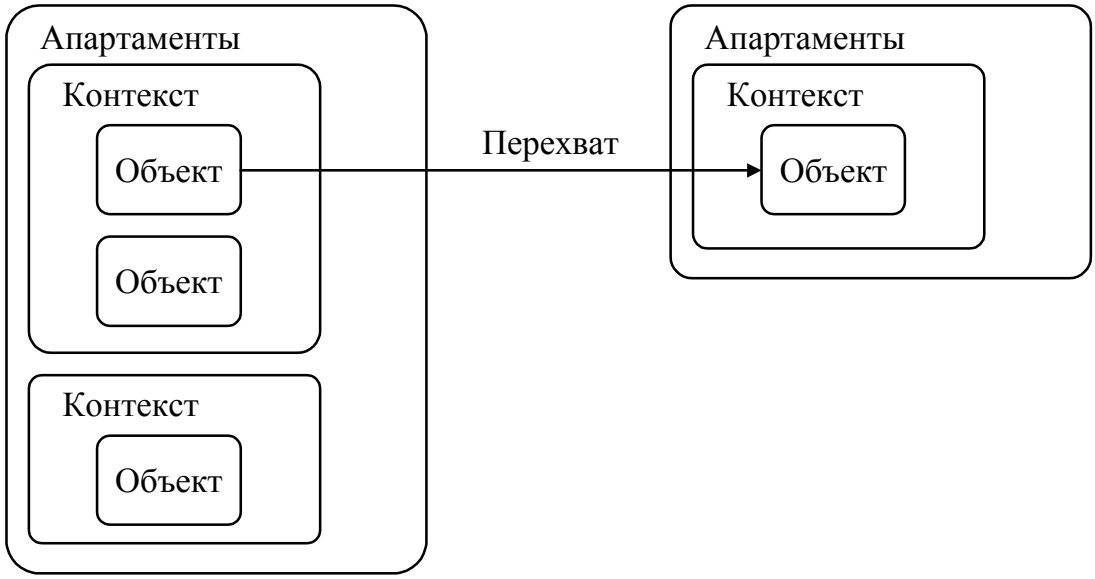


Рисунок 29 — Объект, контекст, апартаменты

Каждый апартамент имеет *контекст по умолчанию*. Когда активизируется несконфигурированный компонент, он связывается с контекстом по умолчанию, который обычно игнорируется.

Контекст на самом деле является абстракцией, объединяющей группу COM-объектов. Он описывает среду объекта, а не его внутренние данные. Сам по себе объект контекста не имеет. Процесс связи объекта с контекстом называется, как было сказано, его активизацией.

Главный вопрос для вновь созданного объекта звучит примерно так: «Как меня будут использовать сейчас?». Поведение объекта определяется его контекстом.

Перед тем, как возможно будет взывать метод объекта, объект создается обычными для COM средствами, а затем активизируется. Далее система COM+ при необходимости выполняет дополнительные действия от имени объекта посредством перехвата вызова метода.

Жизненный цикл объекта показан на рисунке 30.

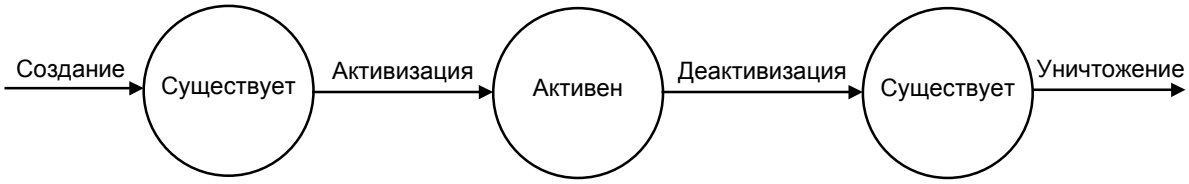


Рисунок 30 — Жизненный цикл объекта COM+

Когда объект активизируется в контексте, вызов его методов из контекста обрабатывается не так, как вызов методов извне. Вызов одного объекта другим в пределах одного контекста не требует вмешательства COM+ и производится непосредственно. Объекты разных контекстов имеют определенную несовместимость их сред выполнения и вызовы, осуществляемые через границы контекстов, требуют перехвата COM- для устранения несовместимости.

Перехватчик представляет собой облегченный прокси, предоставляемый системой COM+ для разрешения несовместимости при вызовах через границы контекстов. Перехватчик выполняет предварительную обработку вызова метода объекта и последующую обработку ответа.

Перехватчик назван облегченным, поскольку он не должен включать изменения потоков. Он выполняет только то, что необходимо для разрешения несовместимости. Обычный же прокси включает в себя перемену потоков и, возможно, перемену процессов.

COM+ обеспечивает также автоматический сервис, который поддерживает пул объектов компонента. Когда объект деактивизируется, он не уничтожается, а попадает в пул созданных объектов. В этом случае его активизация ускоряется, поскольку объект выбирается из пула и связывается с соответствующим контекстом.

Практическая работа №5

В этой работе мы продемонстрируем создание приложения COM+ на основе COM-объекта Beeper.

Прежде всего нужно создать DLL-сервер COM-объекта Beeper.

Создание сервера подробно описано в разделе «Практическая работа №4». Однако, в отличие от приведенного описания, нужно создать COM-объект, кокласс которого является производным от интерфейса IDispatch, а не от интерфейса IUnknown (то есть использовать *дуальный*, а не *custom* интерфейс). Предполагается, что наименование сервера равно BEEPP, наименование кокласса Beeper, и сервер располагается в папке C:\BEEPP (BEEP Plus).

После того, как сервер создан, нужно создать тестирующее приложение BEEPTST. Рекомендуется использовать для этого Visual Basic. Описание тестирующего кода приведено в том же разделе.

Далее нужно создать приложение COM+. Следует открыть «Панель управления», выбрать «Администрирование», «Службы компонентов».

В консоли выберите «Службы компонентов», последовательно раскройте «Компьютеры», «Мой компьютер», «Приложения COM+».

Щелкните правой кнопкой мыши на пункт «Приложения COM+», выберите «Создать приложение». Нажмите кнопку «Далее», затем кнопку «Создать новое приложение». После этого введите имя приложения «BeeperApp», нажмите кнопки «Далее» и «Готово» (рисунок 31).

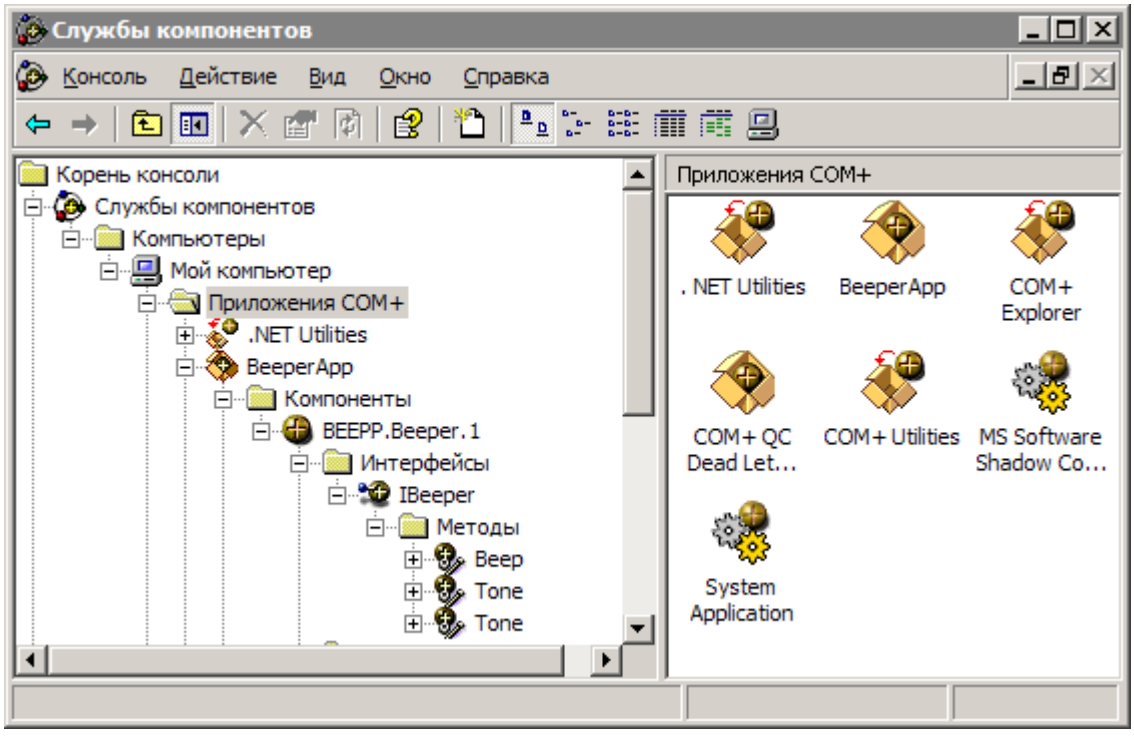


Рисунок 31 — Установка приложения COM+

Чтобы связать приложение с компонентом, щелкните правой кнопкой мыши на раздел «Компоненты» приложения «BeeperApp», выберите «Создать — Компонент», нажмите кнопку «Далее», выберите «Установка новых компонентов», найдите и выберите файл «BEEPPLib.tlb» или файл «BEEPP.dll». Нажимайте кнопки «Далее» до завершения установки. Приложение готово.

Далее требуется тестирующее приложение, аналогичное тому, что использовалось в работе №5. Его нужно скомпилировать для получения исполняемого файла, для чего, например, его нужно запустить на выполнение, в результате которого будет создан исполняемый файл.

Откройте папку, которая содержит файл BEEPTST.exe. Разместите окна на экране таким образом, чтобы одновременно видеть и указанный файл и консоль «Службы компонентов» а в консоли — приложение BeeperApp. Теперь двойным щелчком запустите тестирующее приложение до появления диалога (сообщения о частоте тона). Диалог не закрывайте. Одновременно смотрите на шарик приложения COM+. Наслаждайтесь эффектом вращения шарика COM+, который показывает, что система COM+ находится в действии.

Попробуйте закрыть диалог. Шарик должен продолжать вращаться. Это означает, что объект по-прежнему используется либо находится в пуле объектов. Шарик должен перестать вращаться минут через пять.

Кроме того, открыв диалог атрибутов объекта (рисунок 26, вызывается в контекстном меню как пункт «Свойства»), вы можете попытаться установить те или иные атрибуты. Поскольку работа промежуточного программного обеспечения COM+ прозрачна, в работе объекта вы ничего особенного не заметите.

Дополнительные сведения вы можете найти в главе 15 книги [3].

Напоследок откройте приложение OleView, выберите в меню File - View TypeLib, найдите файл BEEPPLib.tlb, и посмотрите, как выглядит библиотека типов вашего COM-сервера. Для сравнения так же попробуйте открыть файл EXCEL.exe.

Автоматизация (OLE Automation)

Автоматизация (под которой мы будем понимать использование технологии OLE Automation, а точнее — интерфейса IDispatch) выделена в отдельный раздел вследствие ее важности и уникальности.

Важность автоматизации заключается в ее направленности на наиболее полное использование имеющегося на компьютере программного обеспечения, которое предоставляет сервисы в виде COM-объектов.

Уникальность технологии заключается в возможности использовать COM-объекты повсеместно, в том числе в Интернет, в простейших языках типа скриптов, не требующих компиляции, и просто в любых системах программирования, которые поддерживают COM.

Введение

Автоматизация подразумевает программное управление сервисами, которые предоставляются установленным на компьютере программным обеспечением. Главной особенностью автоматизации является отсутствие связи программы, управляющей этими сервисами (называемой контроллером автоматизации), с программными компонентами, являющихся носителями этих сервисов.

Чтобы понять приведенное суждение, нужно хорошо представлять, как происходит так называемое связывание (*linking*) в обычном программировании. Связывание на самом деле есть процесс вычисления адреса (в нашем случае метода объекта). В обычном программировании оно выполняется после процесса компиляции программы, когда станут известны относительные адреса всех функций, используемых клиентской программой. Далее эти адреса записываются в объектный код программы и, таким образом, между программой и всеми используемыми ею функциями (в том числе методами объектов) устанавливается жесткая (на уровне двоичного кода) связь. Это возможно, только если при проектировании клиентского приложения между ним и используемыми другими программными модулями была установлена связь, которая позволяет вызывать функции программных компонентов вне проектируемого компонента. Для примера можно привести диалог References (ссылки), используемый для установления такой связи (рисунок 25).

В автоматизации связь не нужна. Вместо того, чтобы заниматься вычислением адресов, автоматизация использует строки для того, чтобы, во-первых обнаружить наличие COM-объекта, а во-вторых, для того, чтобы убедиться в наличии у него необходимых (требуемых) методов и свойств. Для этой цели и был разработан интерфейс IDispatch.

Говоря образно, контроллер автоматизации сначала «спрашивает» операционную систему «У тебя в системе есть такой-то COM-объект?» и сообщает при помощи строки, какой объект его интересует. В качестве строки выступает уже упоминавшийся ранее ProgID (то есть программный идентификатор COM-объекта).

Если операционная система отвечает, что запрашиваемый сервис существует, одновременно она возвращает указатель на IDispatch. Казалось бы, дальше все просто. Однако именно здесь и начинается самое интересное. Сам по себе интерфейс IDispatch не является представителем функциональности объекта, то есть его *custom* (пользовательским) интерфейсом, он служебный, стандартный. И для всех COM-объектов он один и тот же, одинаков (но только внешне).

Далее контроллер запрашивает у интерфейса IDispatch наличие интересующего его метода примерно так: «У тебя (имеется ввиду COM-объект, сервис) есть такой-то метод?» и сообщает при помощи строки название метода. Примечательно, что название метода в принципе может быть задано на произвольном естественном языке, правда, при этом интерфейс IDispatch должен быть спроектирован подходящим для этого образом.

Если объект «отвечает», что данный метод присутствует, то дальше контроллер должен особым образом упаковать параметры метода и вызвать его через главную «рабочую лошадку» интерфейса IDispatch — его метод Invoke.

Таким образом, при осуществлении связи с объектом через интерфейс IDispatch всегда присутствует некоторая доля неопределенности. Объект может присутствовать в системе, но может и отсутствовать. Метод объекта может быть, а может и не быть. Поэтому при использовании автоматизации всегда нужно проверять результат.

С другой стороны, отсутствие связи позволяет обходиться вообще без компиляции программного кода в скриптах, часто используемых для выполнения различных действий с сервисами, предоставляемыми наличествующим программным обеспечением, а также в Интернет.

Практика является очень наглядным способом демонстрации возможностей той или иной технологии. Предлагаю прямо сейчас продемонстрировать сущность автоматизации на простом примере. К сожалению, невозможно создать из этого примера практическую работу, поскольку для ее выполнения требуется всего лишь от одной до пяти минут времени. Может быть, именно это и позволит показать и важность и уникальность автоматизации как технологии программирования.

Предполагается, что вы выполнили все приведенные в пособии практические работы и у вас есть сервер BEEPP.dll (работа №5).

При необходимости сервер нужно зарегистрировать (например, если вы выполняли работы в компьютерном классе, у вас есть двоичный компонент и прошло некоторое время после вашей работы, в результате чего информация о сервере исчезла из реестра операционной системы).

Теперь нужно создать текстовый файл с именем beep.vbs, который содержит следующий программный код на языке *Visual Basic Scripting Edition*:

```
Dim Q
Set Q = CreateObject("BEEPP.Beeper")
Q.Tone = 70
Q.Beep
```

Теперь запустите этот файл на выполнение, например, дважды щелкните на него мышкой в проводнике Windows. Вы увидите сообщение объекта. И это все…

Если у вас нет сервера BEEPP, то можно, например, использовать *Microsoft Excel*. В этом случае код скрипта должен быть таким:

```
Dim Q
Set Q = CreateObject("Excel.Application")
Q.Workbooks.Add.Worksheets(1).Cells(1, 1) = "Hello, Automation!"
Q.Visible = True
```

Получится демонстрация программы «Hello, World!» в стиле автоматизации.

Автоматизация возможна, если в системе есть серверы. К сожалению, самих серверов не так уж и много, хотя они и покрывают большую часть требуемых возможностей. В первую очередь серверами являются приложения *Microsoft Office*, предоставляющие в ваше распоряжение универсальное средство для работы с текстами *Microsoft Word*, универсальную вычислительную машину *Microsoft Excel* и другие. Из продуктов других производителей отметим универсальное графическое приложение *Autodesk AutoCAD*.

Серверы предлагают для управления так называемую *объектную модель*, представляющую собой связанную группу классов, с помощью которых можно получить доступ ко всем возможностям сервера, доступным при управлении им с помощью пользовательского интерфейса.

Технологии ActiveX являются также частью операционных систем семейства Windows. Поэтому с помощью автоматизации можно управлять и операционной системой. Большая часть ее служб представляет собой сервисы COM-объектов, с помощью которых можно автоматизировать процессы управления.

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

IDispatch

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

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

Изюминкой диспинтерфейса является перенаправление вызовов метода *Invoke* на необходимые методы и свойства пользовательского интерфейса. Перенаправление, или диспетчеризация, дало повод для названия интерфейса.

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

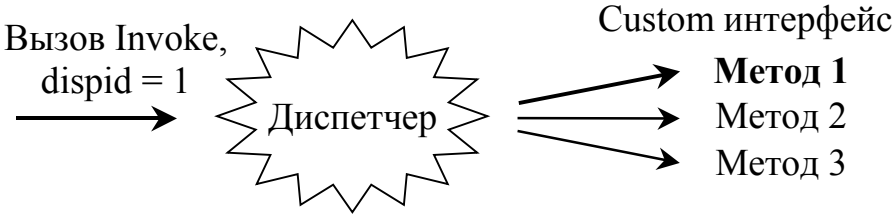


Рисунок 1 — Диспетчеризация вызовов

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

Диспетчер вызовов является частью диспинтерфейса и создается во время проектирования класса автоматизации, поэтому интерфейс дис

петчеризации конкретного класса автоматизации является уникальным, или, говоря иначе, — *все интерфейсы диспетчеризации являются разными*.

Методы диспинтерфейса

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

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

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

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

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

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

riid — зарезервировано, всегда IID_NULL;

rgszNames — указатель на массив имен запрашиваемых методов;

cNames — количество запрашиваемых имен;

lcid — идентификатор локали;

rgDispId — возвращаемый массив *dispid*.

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

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

Метод *Invoke* — главная «рабочая лошадка» диспинтерфейса. Он принимает идентификатор метода или свойства, а при необходимости и параметры метода или новое значение свойства, и формирует запрос к пользовательскому интерфейсу объекта автоматизации. После выполнения запроса метод возвращает значение свойства или результат выполнения метода-функции, а также расширенную информацию об ошибке в случае ее возникновения или номер параметра, имеющего неправильный тип.

Параметры метода *Invoke*:

dispIdMember — *dispid* запрашиваемого метода или свойства;

riid — зарезервировано, всегда IID_NULL;

lcid — идентификатор локали;

wFlags — константа, указывающая на тип запроса.

Значениями могут быть:

DISPATCH_METHOD — запрашивается метод;

DISPATCH_PROPERTYGET — запрашивается чтение свойства;

DISPATCH_PROPERTYPUT — запрашивается запись свойства;

DISPATCH_PROPERTYPUTREF — запись свойства по ссылке;

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

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

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

puArgErr — номер первого параметра метода, имеющего неправильный тип;

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

Упаковка параметров

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

```
typedef struct tagDISPPARAMS {
    VARIANTARG * rgvarg;
    DISPID *     rgdispidNamedArgs;
    UINT         cArgs;
    UINT         cNamedArgs;
} DISPPARAMS;
```

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

rgdispidNamedArgs — массив идентификаторов поименованных параметров;

cArgs — общее количество параметров;

cNamedArgs — количество поименованных аргументов.

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

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

Например, если в языке MSVB определена процедура следующего вида:

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

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

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

Здесь в первом примере используется только позиционная передача, во втором — смешанная, в третьем — только поименованная. Заметим, что при смешанном способе передачи параметров сначала должны быть указаны все позиционные параметры, затем — поименованные.

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

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

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

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

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

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

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

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

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

Примеры вызовов метода Invoke

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

```
#include "stdafx.h"
#include <windows.h>
#include <objbase.h>
#include <oaidl.h>
void main() {
    OLECHAR * szPROGID = L"Excel.Application";
    OLECHAR * szVisible = L"Visible";
    OLECHAR * szEvaluate = L"Evaluate";
    LPDISPATCH pdisp = NULL;
    CLSID clsid;
    VARIANT pVarResult;
    DISPPARAMS dispparamsNoArgs = { NULL, NULL, 0, 0 };
    DISPPARAMS dispparams;
    DISPID dispid;
    DISPID dispidpp[1] = { DISPID_PROPERTYPUT };
    VARIANTARG vararg[1];
    HRESULT hr;
    VariantInit(&pVarResult);
    // Инициализируем OLE
    CoInitialize(0);
    // Получаем CLSID из ProgID
    hr = CLSIDFromProgID(szPROGID, &clsid);
    if (FAILED(hr)) goto error;
    // Создаем экземпляр COM-объекта, запрашиваем IDispatch
    hr = CoCreateInstance(clsid, NULL, CLSCTX_SERVER,
        IID_IDispatch, (void**)&pdisp);
    if (FAILED(hr)) goto error;
    // ЧИТАЕМ СВОЙСТВО Visible
    // Получаем dispid свойства Visible (равный 558)
    hr = pdisp->GetIDsOfNames(IID_NULL, &szVisible, 1,
        LOCALE_USER_DEFAULT, &dispid);
    if (FAILED(hr)) goto error;
    // Метод Invoke
    hr = pdisp->Invoke(dispid, IID_NULL, LOCALE_USER_DEFAULT,
        DISPATCH_PROPERTYGET, &dispparamsNoArgs, &pVarResult, 0, 0);
    if (FAILED(hr)) goto error;
    // Результат в pVarResult должен быть FALSE VT_BOOL
    // УСТАНАВЛИВАЕМ СВОЙСТВО Visible
    VariantInit(&vararg[0]);
    dispparams.rgvarg = vararg;
    dispparams.rgvarg[0].vt = VT_BOOL;
    dispparams.rgvarg[0].boolVal = -1;  // TRUE
    dispparams.cArgs = 1;
```

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

Специальные dispid

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

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

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

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

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

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

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

Существуют также и другие зарезервированные идентификаторы, которые не употребляются или употребляются редко.

Создание объекта автоматизации

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

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

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

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

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

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

Функция *CreateObject* создает объект автоматизации, загружая в память сервер объекта. Если сервер объекта уже загружен, создается еще одна копия сервера. Например, пользователь работает с *Microsoft Excel*, а в то же время контроллер автоматизации требует объект автоматизации этого приложения. Чтобы получить объект, не загружая

сервер, используют функцию *GetObject*. На языке MSVB получить объект от загруженного сервера можно при помощи следующего кода:

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

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

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

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

Связывание

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

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

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

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

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

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

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

Дуальный интерфейс

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

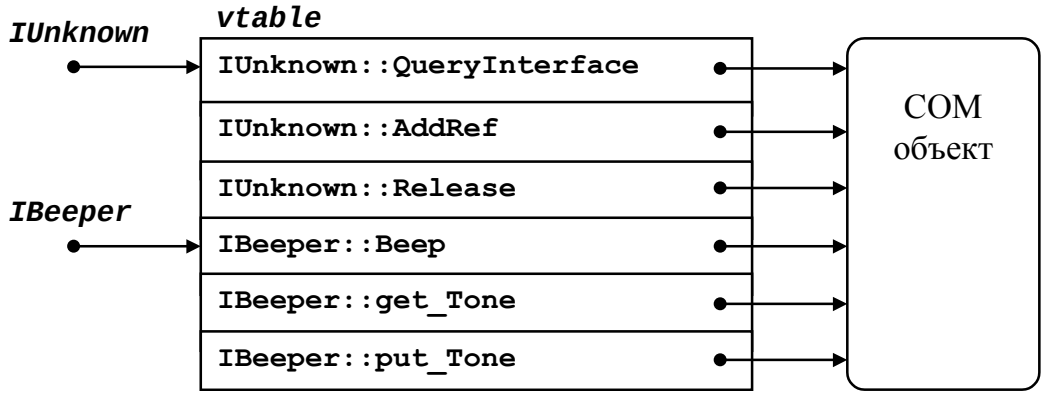


Рисунок 4 — *vtable* для *custom*-интерфейса

В качестве примера здесь используется объект с пользовательским интерфейсом IBeeper, имеющий метод *Beep* и свойство *Tone* (которое реализуется посредством двух функций).

Для автоматизации какого-либо приложения необходимо, чтобы оно экспортировало объектную модель, или как минимум класс, поддерживающий интерфейс диспетчеризации IDispatch.

На рисунке 5 приведена таблица виртуальных методов *vtable* для интерфейса диспетчеризации. Заметим, что интерфейс IBeeper здесь отсутствует — он «спрятан» в диспинтерфейсе.

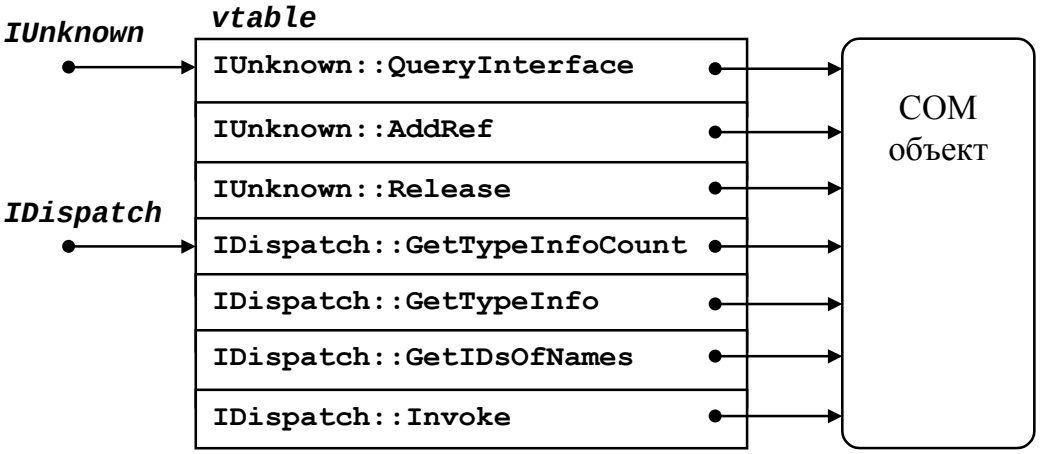


Рисунок 5 — *vtable* для чистого объекта автоматизации

Объект автоматизации, экспортирующий только интерфейс автоматизации, называется *чистым объектом автоматизации*. Для чистого объекта автоматизации можно выполнить только позднее или среднее связывание, что в значительной мере замедляет выполнение вызовов. Функциональность индивидуального объекта в этом случае скрыта, то есть реализована с помощью закрытых методов и свойств класса автоматизации, «спрятанных» на рисунке 5 в прямоугольнике «*Объект*».

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

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

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

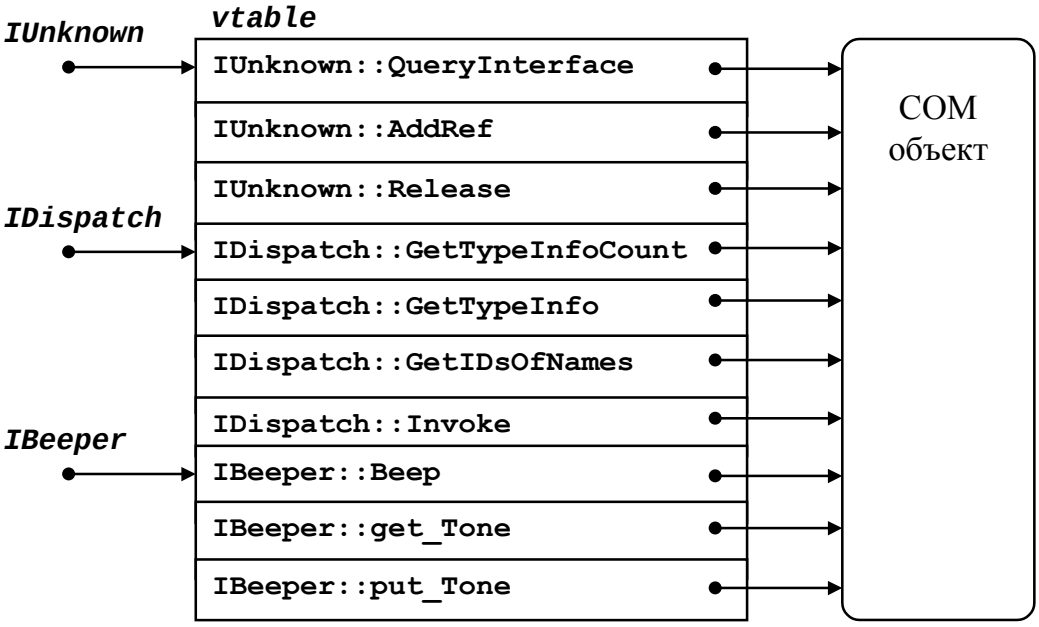


Рисунок 6 — *vtable* для дуального интерфейса

Скрипты

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

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

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

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

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

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

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

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

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

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

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

Автоматизация Microsoft Office

Microsoft Excel

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

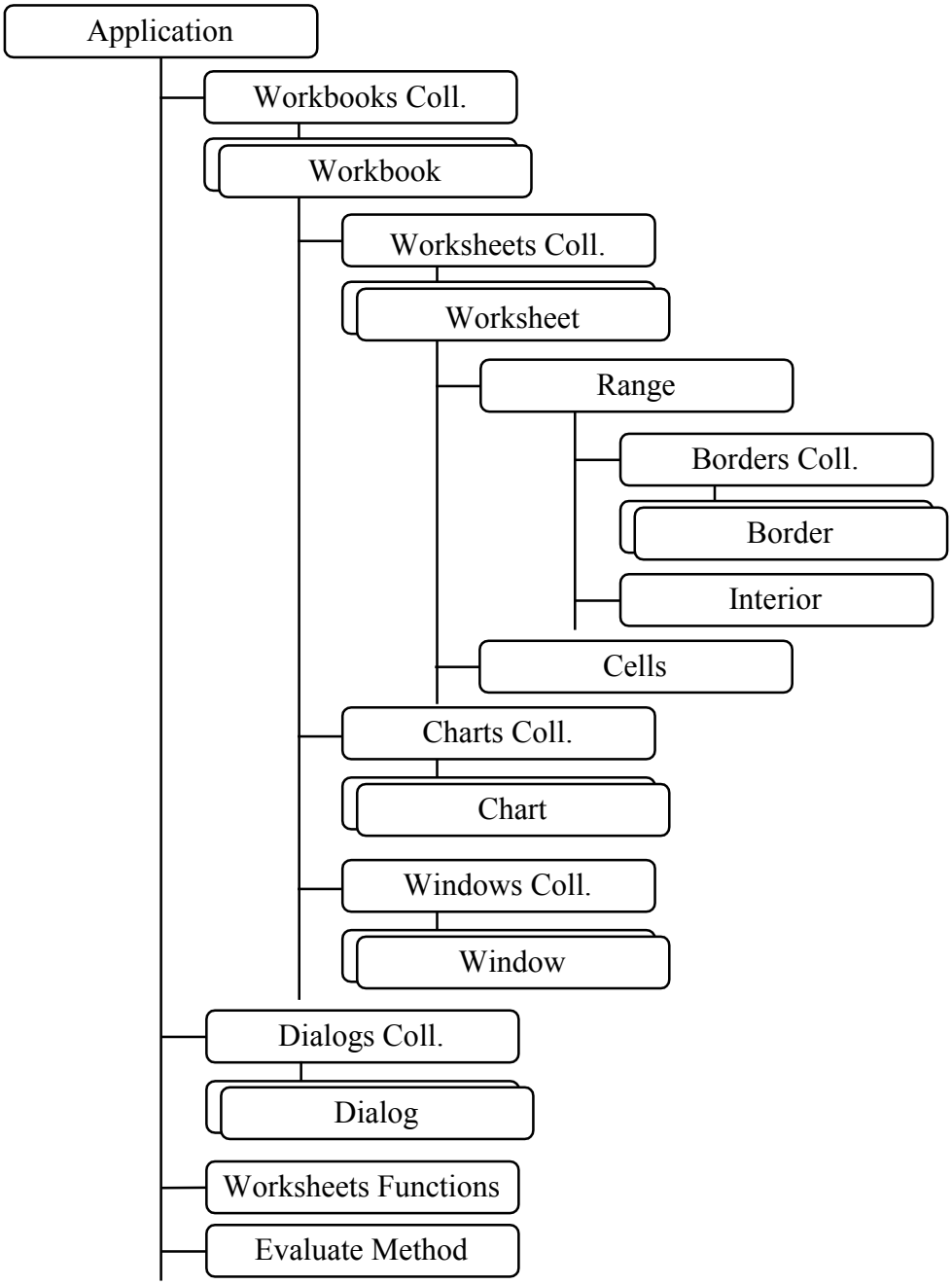


Рисунок 7 — Краткая объектная модель Microsoft Excel

Подробную объектную модель см. справку *Microsoft Excel*.

Express-справка

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

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

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

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

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

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

```
EA.Visible = True
```

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

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

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

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

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

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

Управление приложением

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

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

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

```
EA.DisplayAlerts = False
```

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

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

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

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

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

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

Здесь вместо «выражение» следует подставить выражение. Это может быть либо строковый литерал, либо строковая переменная. Обратим внимание, что результат вычислений всегда следует принимать в переменную типа *Variant*, так как он может содержать нечисловые значения, такие, например, как сообщение об ошибке. Поэтому полученный результат следует проверять, например, так:

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

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

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

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

Управление книгами

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

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

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

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

- `xlWBATWorksheet = -4167 (&HFFFFEFB9)` — расчетный лист;
- `xlWBATChart = -4109 (&HFFFFEFF3)` — лист диаграммы;
- `xlWBATExcel4IntlMacroSheet = 4` — лист макроса;
- `xlWBATExcel4MacroSheet = 3` — лист макроса.

В примере создается новая книга с одним расчетным листом:

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

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

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

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

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

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

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

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

Текстовый файл можно также открыть методом *OpenText*:

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

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

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

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

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

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

При этом каким-либо образом нужно получить спецификацию файла для сохранения. Если книга уже сохранялась, то при использовании метода *SaveAs Microsoft Excel* может показать диалог запроса о перезаписи, на который должен ответить пользователь. Как исключить этот диалог, описано в разделе «Управление приложением». В примере в качестве параметра указана константа, предписывающая сохранить документ как книгу *Microsoft Excel*. Подробнее о параметрах метода см. справку *Microsoft Excel*.

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

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

```
Dim EB As Excel.Workbook
```

```
Set EB = EA.Workbooks(1)
```

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

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

Управление листами

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Работа с ячейками

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

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

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

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

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

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

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

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

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

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

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

Первый индекс массива соответствует строке, второй — столбцу.

Аналогично можно присвоить значение массива диапазону ячеек:

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

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

Оформление ячеек

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

xlDiagonalDown = 5 — диагональная вниз;
xlDiagonalUp = 6 — диагональная вверх;
xlEdgeLeft = 7 — левая;
xlEdgeTop = 8 — верхняя;
xlEdgeBottom = 9 — нижняя;
xlEdgeRight = 10 — правая;
xlInsideVertical = 11 — вертикальная внутри;
xlInsideHorizontal = 12 — горизонтальная внутри.

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

xlContinuous = **1** — сплошная;
xlDash = **-4115** (**&HFFFFFEFD**) — штриховая;
xlDashDot = **4** — штрих пунктирная;
xlDashDotDot = **5** — штрих-два пунктира;
xlDot = **-4118** (**&HFFFFFEFA**) — пунктирная;
xlDouble = **-4119** (**&HFFFFFEF9**) — двойная;
xlLineStyleNone = **-4142** (**&HFFFFFEFD2**) — отсутствует;
xlSlantDashDot = **13** — наклонный штрих-пунктир.

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

xlHairline = **1** — очень тонкая;
xlThin = **2** — тонкая;
xlThick = **4** — толстая;
xlMedium = **-4138** (**&HFFFFFEFD6**) — средняя.

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

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

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

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

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

Microsoft Word

Microsoft Word представляет собой универсальный программный компонент для обработки текстов. Значительно сокращенная объектная модель этого приложения приведена на рисунке 8.

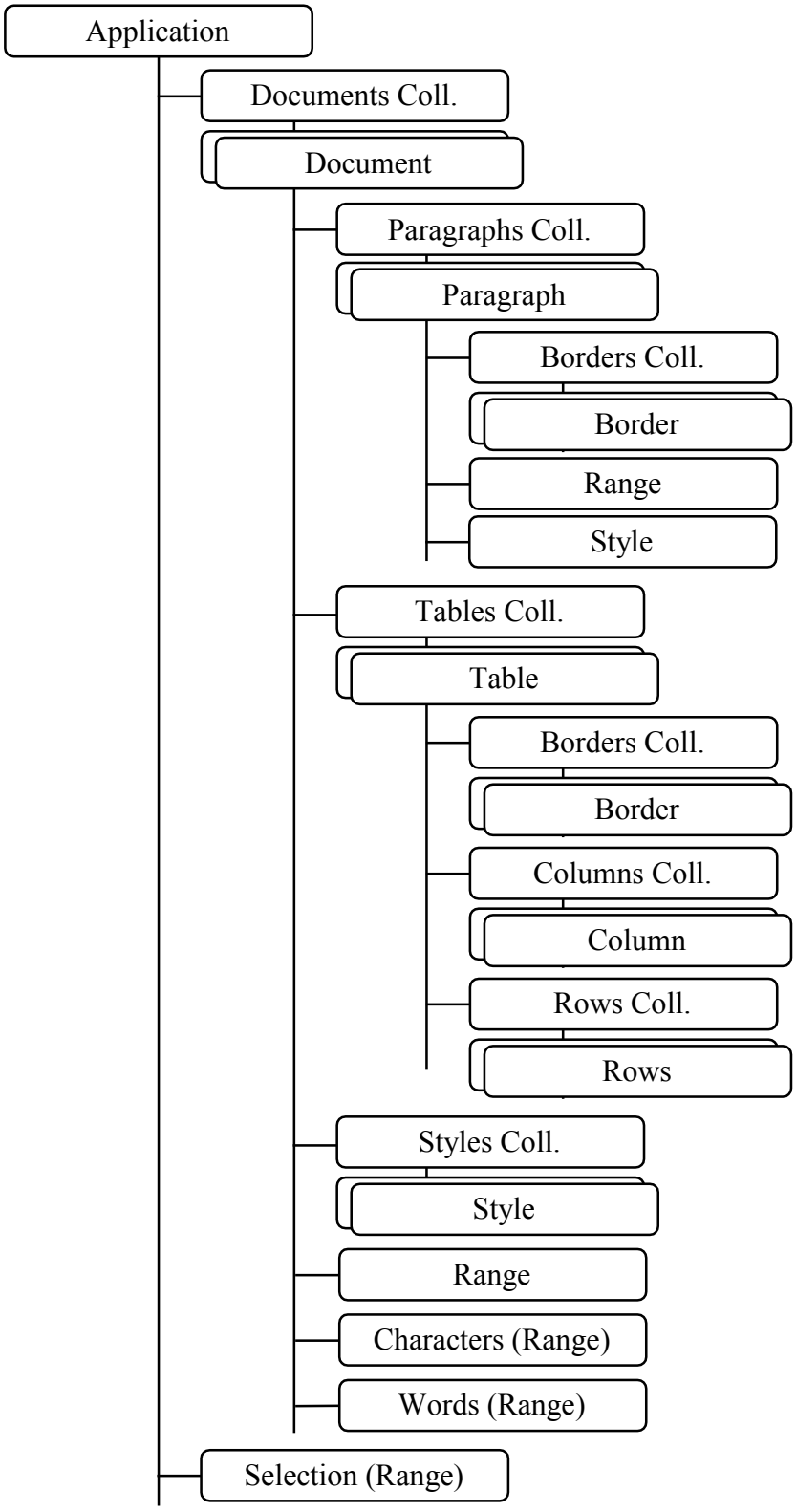


Рисунок 8 — Краткая объектная модель Microsoft Word

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

Основными элементами объектной модели *Microsoft Word* являются объект *Document*, представляющий документ, объект *Selection*, представляющий выделенную часть документа, объект *Range*, представляющий часть документа, выбранную для выполнения действий. Объект *Selection*, и некоторые другие, например *Characters* и *Words*, возвращают объект *Range*.

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

Управление приложением

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

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

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

```
WA.Visible = True
```

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

wdAlertsNone = 0 — отключить диалоги и сообщения об ошибках;

wdAlertsMessageBox = -2 — отключить диалоги;

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

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

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

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

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

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

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

Управление документами

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

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

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

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

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

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

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

Формат документа задается одной из следующих констант:

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

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

```
WD.SaveAs Path
```

Метод имеет десять дополнительных параметров сохранения.

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

Закрыть приложение можно при помощи метода *Quit*.

Управление текстом

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

В большинстве случаев для выполнения действий с текстом требуется выделить или выбрать его часть. Для этой цели используется объект *Range*, представляющий собой непрерывный блок документа. Объект *Range* имеет два свойства, которые указывают на начало (свойство *Start*) и конец (свойство *End*) блока. В следующем примере показано, как выделить второй по порядку абзац текста (включая маркер конца абзаца):

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

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

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

- свойства *Bold* и *Italic* управляют начертанием «полужирный» и «курсивный» соответственно;
- свойство *Case* управляет регистром букв. Это свойство может принимать следующие значения:

`wdLowerCase` = `0` — перевести в нижний регистр;
`wdUpperCase` = `1` — ПЕРЕВЕСТИ В ВЕРХНИЙ РЕГИСТР;
`wdTitleWord` = `2` — Каждое Слово С Прописной Буквы;
`wdTitleSentence` = `4` — Предложение с прописной буквы;
`wdToggleCase` = `5` — переключить регистр на противоположный.

- свойство *Characters* возвращает коллекцию одиночных символов;
- свойство *Font* возвращает объект *Font*, с помощью которого можно управлять всеми параметрами шрифта — гарнитурой, размером, начертанием, подчеркиванием и т.п.;
- свойство *LanguageID* управляет языком. Значение свойства равно идентификатору языка операционной системы. Например, русский язык имеет идентификатор 1049, английский — 1033;
- свойство *Style* управляет стилем. Значением этого свойства является название стиля, зарезервированная константа (для встроенных стилей) или объект *Style*;
- свойство *Text* возвращает или устанавливает чистый текст;
- свойство *Words* возвращает коллекцию слов.

Некоторые методы объекта *Range*:

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

- метод *InsertParagraph* заменяет блок новым параграфом;
- метод *Copy* копирует блок в буфер обмена;
- метод *Paste* вставляет блок из буфера обмена;

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

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

```
WD.Paragraphs.Add
```

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

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

Поиск и замена

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

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

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

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

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

Проверка правописания

Для проверки правильности написания слов *Microsoft Word* использует словари. Слово сравнивается с имеющимися в словаре и, если слово в словаре не обнаружено, высказывается предположение о правильном написании. Если слово не обнаружено в основном словаре, оно проверяется в дополнительных словарях (до десяти словарей можно указать в параметрах метода). Кроме проверки правописания *Microsoft Word* проверяет также грамматику.

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

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

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

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

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

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

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

Дополнительные материалы

Строковые типы Windows

Символы

Для представления символьных и строковых данных в системе Windows используется множество типов, макросов и функций. Типами для символьных данных являются *узкие* и *широкие* (*wide*) символы. Узкий символ занимает в памяти один байт, а широкий — два.

Узкий символ — это стандартный тип *char*.

Широкий символ определяется как

```
typedef unsigned short wchar_t;
```

то есть как беззнаковое 16-битное целое (файл *ctype.h*).

При этом неявно подразумевается, что тип *wchar_t* предназначен для кодировки *Unicode* (UTF-8), в которой для представления символов как раз используется 16 бит.

Основной причиной возникновения множества типов для представления строк является наличие множества кодировок для представления самих символов. Так, первоначально подавляющее большинство платформ (операционных сред) использовали для представления строк таблицы кодировок ASCII, что почти равнозначно таблицам кодировок ANSI. Для представления одного символа в этих таблицах используется *один байт*. Язык программирования *C* внес некоторую путаницу в представление символов, определив символ как *знаковое* 8-битное целое. Это означает, что символы из второй половины таблицы кодировки формально имеют отрицательные коды ASCII.

Одновременно для представления отдельных символов была разработана система *Unicode*, в которой символ кодируется 16-битным беззнаковым целым числом. Цель *Unicode* — обеспечить кодировку всех возможных символов, включая китайский, японский и корейский алфавиты, насчитывающие до 5 тысяч знаков.

Кодировка *Unicode*, однако, не поддерживалась большинством платформ. Для кодировки символов японского, китайского и корейского алфавитов были предложены различные способы записи, основанные на однобайтном представлении (на кодировке ANSI). Эти способы получили название DBCS (*double-byte character set* — двухбайтный набор символов) и MBCS (*multi-byte character set* — многобайтный набор символов). В принципе, DBCS и MBCS — это одно и то же. Проблема с подобными наборами заключается в том, что некоторые символы в них являются *лидирующими*, то есть используются для идентификации одного или нескольких последующих байт. Эти проблемы, однако, нас не каса

ются, и в дальнейшем мы не будем рассматривать наборы типа DBCS. Они просто создают еще большую путаницу.

Итак, символы могут быть *узкими*, то есть представляемыми при помощи одного байта, и *широкими*, то есть представляемыми при помощи двух байт.

Платформы

Платформа — это специфичное операционное окружение, внутри которого разрабатываются и функционируют программы. Если говорить о платформах Windows, то существуют платформы *Win16*, *Win32* и *WinNT* (и это не все), которые определяют установки по умолчанию, форматы файлов, средства построения результирующих файлов и т.п. Так, платформа *Win16*, на которой построена операционная система *Windows 3.x*, определяет, что строки — это массивы символов ANSI, и символы *Unicode* для этой платформы являются неизвестным понятием. Системные функции этой платформы не предназначены для работы со строками *Unicode*. Платформа *Win32* поддерживает *Unicode* в каком-то неполном объеме, и является переходной к платформе *WinNT*, в которой поддержка *Unicode* полная. Что означает «полная» и «неполная» поддержка, точно неизвестно. Замечу только, что файловые системы Windows, начиная с версии 98-2, поддерживают *Unicode* для именования файлов и каталогов (папок).

Обеспечение *межплатформенной совместимости* является одной из важнейших черт средств программирования. Для этой цели в Windows введен еще один тип символов, который является совместимым с любой платформой — тип _TCHAR (TCHAR). Во время компиляции программы он определяется как однобайтный или двухбайтный символ в зависимости от платформы, для которой строится программа. Для этого типа введено понятие *символа общего назначения — Generic char*.

Для различения платформ *Win16* и *Win32* введены типы: *CHAR* — это *char* для *Win32*, а *WCHAR* — это *wchar_t* для *Win32*. Возникновение технологий OLE и COM также внесло свою лепту в путаницу с символами. Изначально технология COM предполагала использование только кодировок *Unicode*. Для обеспечения совместимости с OLE в Windows определен тип OLECHAR, который, понятно, есть *wchar_t*.

Появление платформы .NET несколько улучшило ситуацию (типа все есть *Unicode*), однако также внесло свою долю путаницы. Например, теперь есть функции типа *wsprintf*, имеющие на конце дополнительные буквы A или W, означающие ANSI или Wide символы, то есть результирующие строки различного типа.

Строки

Строка — это массив знаков, определяемый своим *указателем*. Наиболее известен строковый тип *char**, в котором конец строки определяется нулевым знаком — *null-terminated string*.

Такое представление строки получило распространение в программировании в связи с языком *C* и внесло одну важную проблему, которая легла на плечи программиста — выделение и освобождение памяти. В отличие от обычной переменной, строковая не является указателем на ячейку или ячейки памяти, в которых находится объект. Вместо этого строковая переменная — это указатель на указатель на знаковый тип. Можно, например, так задать строковый тип:

```
typedef char * string;
```

Объявление переменной строкового типа не определяет никакого объекта, который можно использовать прямо, как в случае с переменной целого типа, которой, к примеру, можно сразу присвоить какое-нибудь значение:

```
long j;
j = 100000;
```

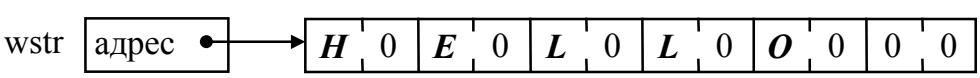
Вместо этого в обязанность программиста вводится предварительное выделение памяти при помощи конструктора *new* или функции *malloc*. Если использовать *C++*, то выделить память можно так:

```
char *str;
str = new char[6];
```

Если строковое значение этого массива по ходу программы должно изменяться в своей длине, то программист обязан перераспределять память, освобождая выделенную первоначально и требуя новую. После завершения работы со строковым объектом, программист должен освободить память при помощи деструктора *delete*[] или функции *free*, например:

```
delete[] str;
```

В принципе, строка, которая содержит не узкие, а широкие символы, принципиально ничем не отличается от строки, завершающейся нулём. Вот пример для того же слова *Hello* в *Unicode* исполнении:

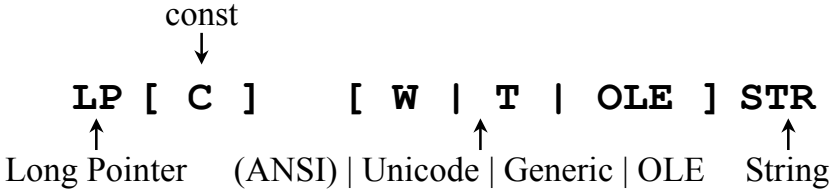


Не единственная проблема, которая добавляется в связи с использованием *Unicode*, заключается в необходимости преобразования строки *Unicode* в строку ANSI и обратно. Учитывая, что в большинстве случаев компьютер настроен на одновременное отображение стандартных символов ANSI и национальных символов, такое преобразование не приводит к заметной для пользователя потере символов.

Для *C*-программиста, однако, возникают проблемы с приведением типов и преобразованием строк. Если в стандартном *C* единственными функциями сравнения строк были *strcmp* и *strcmpi*, то множество знаковых типов в Windows породило соответствующее множество строковых, то есть указательных типов и, соответственно, множество функций для работы с ними. Стремление поддерживать старые форматы данных, библиотеки и системы программирования привело к тому, что в одной программе одновременно используются сразу несколько строковых типов. Мало того, что они могут оказаться несовместимыми по объему занимаемой памяти (узкими или широкими), необходимо еще обеспечить совместимость на уровне компилятора, на момент контроля типов. Две строки, которые кажутся программисту совершенно одинаковыми и фактически совершенно одинаково размещены в памяти, компилятор считает несовместимыми, если они имеют разные указательные типы. В большинстве случаев проблема несовместимости в этом случае решается за счет явного приведения типа, как в следующем примере кода:

```
#include <windows.h>
#include <tchar.h>
void main(void) {
   char * str_a;
   LPCTSTR str_t = _T("abc");
   str_a = (char *)str_t;
}
```

Для определения строковых типов в *Windows* используются определения типов *typedef*, начинающиеся с LP — *Long Pointer*. Следующий рисунок поясняет, как определить тип указателя:



Так, например, LPCSTR — это постоянная строка ANSI, то есть тип *const char* *, а LPOLESTR — это указатель на строку OLE, то есть *OLECHAR* *, или *wchar_t* *.

Для перевода строк из формата *Unicode* в MBCS и обратно используются макросы, доступные после объявления USES_CONVERSION.

```
A2CW     (LPCSTR)     -> (LPCWSTR)
A2W      (LPCSTR)     -> (LPWSTR)
W2CA     (LPCWSTR)    -> (LPCSTR)
W2A      (LPCWSTR)    -> (LPSTR)
```

Для перевода строк OLE используются следующие макросы:

```
T2COLE   (LPCTSTR)    -> (LPCOLESTR)
T2OLE    (LPCTSTR)    -> (LPOLESTR)
OLE2CT   (LPCOLESTR)  -> (LPCTSTR)
OLE2T    (LPCOLESTR)  -> (LPCSTR)
```

При необходимости эти макросы выполняют преобразование типов при помощи системных функций, для этой цели предназначенных, а именно при помощи основных функций *MultiByteToWideChar* и *WideCharToMultiByte*. Ничто не мешает использовать эти функции прямо, если помнить, какие параметры (до восьми) требуются. *MultiByte* в данных функциях означает, как ни странно, однобайтные наборы (ANSI). Впрочем, объяснить это достаточно просто, если вспомнить, что многобайтные кодировки основаны на ANSI.

Для обычных операций со строками теперь следует использовать модернизированные версии строковых функций *strcpy*, *strcmp*, *strlen* и других. Их так много, что одно только перечисление займет целый лист. Вот примеры только некоторых из них:

wcscmp – сравнивает строки с широкими символами

_mbscmp – сравнивает строки с многобайтными символами

wcscpy – копирует строку широких символов

_mbscpy – копирует строку многобайтных символов

wcscat – объединяет две строки широких символов

_mbscat – объединяет две строки многобайтных символов

wcstombs – переводит широкие символы в узкие

mbstowcs – переводит узкие символы в широкие

wcsrchr – сканирует строку широких символов и определяет вхождение заданного символа

_mbsrchr – сканирует строку многобайтных символов и определяет вхождение заданного символа

Еще одна проблема со строками в OLE — определение строк символами *Unicode*. Для этой цели используется непривычная форма с использованием буквы *L*:

```
OLECHAR* pOLEStr;
pOLEStr = L"Hello";
```

Однако, такой прием может привести к проблемам на платформах, не поддерживающих *Unicode*. Поэтому существует макрос OLESTR, который определяется по-разному в зависимости от платформы:

```
pOLEStr = OLESTR("Hello")
```

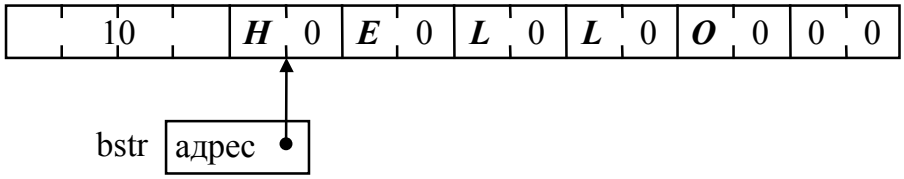
К сожалению, это не все. Есть еще два макроса для определения строк, а именно, _T и _Text, которые абсолютно идентичны. Они определяют строковые литералы общего строкового типа, который будет представлен либо узкими, либо широкими символами, в зависимости от платформы.

Напоследок приводится таблица основных типов.

Тип	*Описание*
char	8-битный знаковый символ (символ ANSI)
wchar_t	16-битное беззнаковое целое (символ Unicode)
CHAR	Версия char для Win32
WCHAR	Версия wchar_t для Win32
OLECHAR	Версия wchar_t для OLE
_TCHAR	Символ общего типа — char или wchar_t
LPSTR, LPCSTR	Указатель на символ для Win32
LPWSTR, LPCWSTR	Указатель на широкий символ для Win32
LPOLESTR, LPCOLESTR	Указатель на широкий символ для OLE
LPTSTR, LPCTSTR	Указатель на символ общего типа для Win32
_T(str), _TEXT(str)	Макросы для формирования строк общего типа
OLESTR(str)	Макрос для формирования строки общего типа

Тип автоматизации BSTR

Проблема с OLE строками заключается в том, что не все языки программирования понимают строки, заканчивающиеся нулем. Языки типа *Visual Basic*, *Java*, *VBScript* и *JavaScript* предполагают, что строки имеют фиксированную длину. Для совместимости с этими языками COM (или правильнее сказать OLE) вводит еще один строковый тип — BSTR (*Basic STRing*). Объяснить, что это такое, проще всего при помощи рисунка:



Из рисунка следует, что это *Unicode*-строка, предваряемая четырехбайтным счетчиком символов. Особенностью этого типа является также то, что в конце этой строки обязательно присутствует нулевой символ,

даже если вы его и не определяли. Это позволяет простым образом преобразовать BSTR к типу, например, LPOLESTR. Однако, не все так просто. Дело в том, что поскольку длина строки зафиксирована в специальном счетчике, сама строка может содержать нулевые символы как часть строки, а не как завершающие. Поэтому, преобразование BSTR к любому другому Windows строковому типу следует выполнять с осторожностью.

С точки зрения программирования, работа с типом BSTR непосредственным образом представляет значительные трудности. Например, чтобы получить новую BSTR, нужно выделить память размером с длину строки, умноженную на два плюс 6, и к полученному указателю прибавить 4. Вряд ли это вызовет энтузиазм у большинства программистов. Поэтому вместе с появлением типа BSTR в API Windows были добавлены соответствующие системные функции, начинающиеся с *Sys*:

SysAllocString – определяет строку,

SysReAllocString – переопределяет строку,

SysFreeString – освобождает память, выделенную под строку,

и другие…

Для упрощения работы с BSTR библиотека ATL предлагает также упаковочный класс (*wrapper class*) *CComBSTR*. С его помощью, например, можно выполнить простое присваивание одной строки другой.

Фактическое определение BSTR выглядит так:

```
typedef wchar_t * BSTR;
```

Следующий пример кода показывает приемы работы с BSTR:

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

Тип автоматизации VARIANT

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

VARIANT — это структура, которая может хранить значение любого типа, совместимого с автоматизацией. Тип, совместимый с автоматизацией — на самом деле тип, который может свободно использоваться не только в автоматизации, но и вообще в программировании. При этом гарантируется его правильная интерпретация и преобразование в другой тип, совместимый с автоматизацией.

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

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

Структура VARIANT имеет размер 16 байт. Значения, размер которых не превышает 8 байт, записываются непосредственно в структуру.

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

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

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

VT_EMPTY — указывает на отсутствие значения (пусто);

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

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

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

Byte — беззнаковое целое число (1 байт);

Integer — знаковое целое число (2 байта);

Long — знаковое целое число (4 байта);

Single — вещественное число одинарной точности (4 байта);

Double — вещественное число двойной точности (8 байт);

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

Currency — точное вещественное число (8 байт);

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

Variant — значение типа VARIANT;

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

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

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

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

Для работы с типом VARIANT следует использовать системные функции. Перечень и описание функций здесь не приводится.

Порядок использования типа VARIANT следующий:

1) инициализировать (*VariantInit*);

2) записать тип данных в поле *vt*;

3) записать значение в соответствующее типу поле объединения;

4) выполнить операции;

5) очистить переменную (*VariantClear*).

Последнее действие особенно важно, когда вариант хранит нечисловое значение, т.к. в этом случае значение хранится не в структуре.

Следующий пример поясняет сказанное:

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

Для выполнения математических и логических операций с переменными типа VARIANT также используют системные функции.

Объектная файловая модель

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

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

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

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

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

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

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

Диски

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

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

Тип диска определяется одной из следующих констант:

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

Важными свойствами объекта *Drive* являются также:

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

Каталоги

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

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

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

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

Важными свойствами объекта *Folder* являются:

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

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

Normal = **0**
ReadOnly = **1**
Hidden = **2**
System = **4**
Volume = **8**
Directory = **16**
Archive = **32**
Compressed = **2048**
Alias = **1024**

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

Файлы

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

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

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

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

Объект *File* обладает следующими важными свойствами:

Attributes — атрибуты файла (описаны выше);
Drive — объект *Drive*, описывающий содержащий файл диск;
Name — название файла;

ParentFolder — возвращает объект *Folder* родительского каталога;
Path — спецификация файла (свойство по умолчанию);
Size — размер файла.

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

Потоки

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

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

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

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

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

Важные методы объекта *TextStream*:

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

Свойства объекта *TextStream*:

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

В следующем примере показано, как открыть файл в виде потока и построчно вывести его в окно отладки *Immediate* (в среде MSVB или VBA):

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

В следующем примере тот же файл читается целиком в строковую переменную, которая и выводится в окно *Immediate* (в среде MSVB или VBA):

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

Рекомендуемая литература

1. Бокс Д. Сущность технологии COM. Библиотека программиста. — СПб.: Питер, 2001. — 400 с.: ил.

2. Коберниченко Алексей. Visual Studio 6. Искусство программирования. — М.: «Нолидж», 1999. — 256 с., ил.

3. Оберг, Роберт, Дж. Технология COM+. Основы и программирование.: Уч. пос. — М.: Издательский дом «Вильямс», 2000. — 480 с.: ил. — Парал. тит. англ.

4. Трельсен Э. Модель COM и применение ATL 3.0: Пер. с англ. — СПб.: BHV — Санкт—Петербург, 2000. — 928 с.: ил.

Владимир Вадимович Пономарев
Введение в ActiveX
Учебно-методическое пособие

Отпечатано с готового оригинал-макета
Издательство ОТИ НИЯУ МИФИ, 2012
Тираж 40 экз.