МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

Введение в ActiveX

Часть 1. Инфраструктура COM

Учебно-методическое пособие
по дисциплине
«Современные технологии программирования»

Озерск — 2016

УДК 681.3.06
П56

Пономарев В.В. Введение в ActiveX. Часть 1. Инфраструктура COM. Учебно-методическое пособие по дисциплине «Современные технологии программирования».
Озерск: ОТИ НИЯУ МИФИ, 2016. — 82 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)
2)

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

ОТИ НИЯУ МИФИ, 2016

Содержание

Обозначения и сокращения

ATL	ActiveX Template Library (библиотека шаблонов ActiveX)
COM	Component Object Model (модель компонентных объектов)
LRPC	Lightweight RPC
MSVB	Microsoft Visual Basic
MSVS	Microsoft Visual Studio
OLE	Object Linking and Embedding (связь и внедрение объектов)
RPC	Remote Procedure Call (вызов удаленной процедуры)
VBA	Visual Basic for Application
VBScript	Visual Basic Scripting Edition

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. Изучение технологий *ActiveX* представляется логическим продолжением изучения концепций объектно-ориентированного программирования, и предтечей изучения .NET.

Пособие состоит из двух частей.

В первой части излагаются основы COM.

Вторая часть посвящена технологии OLE Automation.

В каждой части есть описания практических работ, которые требуется выполнить для закрепления теоретического материала, а также вопросы для самоконтроля.

Первая часть состоит из четырех глав, — «Понятие о COM», «Инфраструктура COM», «Использование ATL 3.0», «COM+».

В главе «Понятие о 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 и клиента COM средствами ATL3.

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

В конце части приводятся литературные источники, рекомендуемые для более детального изучения технологий.

Понятие о COM

Проблемы программирования

Одной из самых важных и сложных задач программирования как отрасли производства является повторное использование наработанного за годы практического программирования кода (*code reuse*).

На протяжении многих десятилетий программы в большинстве случаев создаются «с нуля», в то время как во всех других отраслях промышленного производства используются те или иные *строительные компоненты*.

Производители компьютеров используют, например, микросхемы, транзисторы, конденсаторы и другие радиоэлектронные компоненты. Строители используют кирпичи, блоки, панели, балки и еще много чего. Машиностроители используют болты, гайки, шарикоподшипники и т.п.

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 него получить кремниевую пластину, на ней сформировать *p-n* переходы, из них выстроить логические схемы…, и так до тех пор, пока, наконец, не будут получены вычислительные устройства, из которых можно создать компьютер.

Нельзя сказать, что в программировании все уж так плохо.

Существует множество технологий, методологий и сред проектирования, которые в значительной степени помогают быстро создавать качественное программное обеспечение. Здесь можно отметить объектно-ориентированное программирование, шаблоны, паттерны проектирования, CASE средства на основе UML 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 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 классов срабатывает так называемый эффект *NIH* (*not invented here*, изобретено не здесь).

В результате готовые тексты подвергаются модернизации, приспосабливанию к условиям конкретного проекта. Это ведет к порождению ошибок, затратам времени на их обнаружение, устранение, отладку и сопряжение модулей.

Существуют также проблемы с библиотеками программ, целью которых как раз и является повторное использование. Однако невозможно, например, взять математическую библиотеку FORTRAN, и присоединить ее к проекту, написанному на Паскале. Поэтому создаются новые библиотеки, которые можно использовать только в определенном контексте, в определенной среде и определенном языке программирования.

Целью COM является создание «*неразборных*» двоичных модулей многократного использования, применимых в любой операционной среде, в сочетании с любым языком программирования, при условии, что среда поддерживает инфраструктуру COM.

Документ как центр системы

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 отражает идеологическую направленность *Windows* на документ, как на главное, основное понятие в работе пользователя.

Понятие документа связывается с приложением. Документ, — это те данные, с которыми работает конкретное приложение.

Например, для программы *Paint* данными являются растровые картинки типа bmp, gif, png. Для программы *Notepad* (*Блокнот*) данными является текстовые файлы типа txt.

С другой стороны, понятие документа связано с информацией.

Например, информация в файле типа bmp или png может быть обработана приложением *Paint*, а информация в файле типа txt может быть обработана приложением *Notepad*.

Связь между документом и информацией однозначная. Документ типа bmp содержит описание растровой картинки. Связь между приложением и документом не однозначная. Документ типа bmp может быть обработан приложениями *Paint* и *Photoshop*. Аналогично документ типа txt может быть обработан приложениями *Notepad* и *WordPad*.

С этой точки зрения все операционные системы условно можно поделить на системы, ориентированные на приложение (*application-centric operating system*), и ориентированные на документ (*document-centric operating system*).

В системах, ориентированных на приложение, приложение является центральным понятием, потому что в них пользователь сначала открывает программу, а затем с его помощью открывает документ. Обычно это системы, представляющие пользовательский интерфейс в виде командного языка. В некоторых случаях это находит отражение в самом командном языке в виде команды `run` (запустить программу).

В системах, ориентированных на документ, пользователь «открывает» документ для работы с ним, вместо того, чтобы открывать приложение. Операционная система запускает приложение в соответствии с тем, документ какого типа требуется открыть. Это отражается также в изменении терминологии. Если для запуска программы используется команда `run`, то для «запуска» документа используется термин «*открыть*». В системах, ориентированных на документ, термин «*открыть*» применяется также и к приложению.

Представителем системы, ориентированной на приложение, является MS-DOS. Система *Windows* является системой, ориентированной на документ. Предлагаемый ей способ работы с информацией более точно отражает практический опыт пользователя.

Составной документ

Другим отличием *Windows* от MS-DOS является многозадачность.

Многозадачность позволяет пользователю открыть и работать одновременно с несколькими документами. При этом появляется необходимость соединения информации разного рода в одном документе.

В MS-DOS пользователь в один момент времени работает только в одной программе. Предположим, он редактирует текстовый документ. В какой-то момент ему требуется вставить в документ картинку. Сделать этого он не может, картинки не обрабатываются текстовыми редакторами. И взять готовую картинку он не может, для этого ему нужно вернуться в MS-DOS. Кроме того, документы MS-DOS не предназначены для хранения информации разного рода. Если документ — растровая картинка, то кроме картинки в документ ничего записать нельзя.

Соединение информации разного рода в одном файле документа в MS-DOS не невозможная, но сложная, и не системная задача.

С появлением системы *Windows* необходимость соединения информации разного рода в одном документе стала объективной реальностью и превратилась в общесистемную задачу.

Появляется понятие составного документа (*compound document*).

Составной документ состоит из информации определенного рода, которая для данного типа документа является *основной*. Любая *инородная* информация вставляется в составной документ в виде *объекта*.

В качестве примера будем рассматривать документ приложения *Word*, основную информацию которого составляет *текст*, и документ приложения *Paint*, основную информацию которого составляет растровое изображение, *картинка* (рисунок 1).

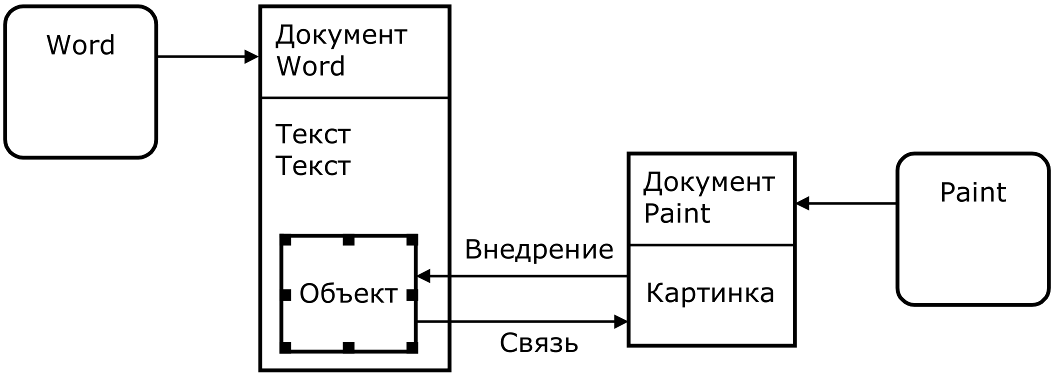


Рисунок 1 — Составной документ

На рисунке в текст документа *Word* вставлена картинка, созданная приложением *Paint*. Картинка по отношению к тексту является внедренным объектом. Объекты в документе *Word* легко найти. При выделении объекта вокруг него появляется рамка с объектными маркерами.

OLE

Чтобы вставить инородную информацию в составной документ, в *Windows* был разработан механизм *Object Linking and Embedding* (OLE, читается *olay*), *связь и внедрение объектов*. Он включают в себя:

- *буфер обмена* (*clipboard*),
- операции *копировать*, *вырезать*, *вставить* (*copy*, *cut*, *paste*),
- механизм *внедрения* объекта в составной документ, и
- механизм *связи* объекта с исходным документом.

Идея заключается в том, что информацию некоторого рода обрабатывает приложение, которое для этого предназначено. Текст составного документа *Word* обрабатывает *Word*, а картинку в тексте, то есть объект, обрабатывает *Paint*. Чтобы изменить внедренную картинку, нужно, например, дважды щелкнуть на объект. При этом откроется приложение *Paint*, в котором картинку можно изменить.

В ранних версиях *Word* приложение *Paint* открывалось в окне *Word*, а в меню *Word* появлялся пункт «*Закрыть и вернуться в Word*».

Объект всегда *внедряется* (*embed*). Однако можно установить связь с исходным документом объекта, если использовать диалог *Специальная вставка*. В этом диалоге для внедряемого объекта можно установить флажок «*Связать*» (*Link*). Тогда изменения, которые вносятся в исходный документ объекта, отображаются в составном документе.

На самом деле каждый раз когда, работая в *Windows*, вы что-то копируете и затем куда-то вставляете, срабатывает OLE. Если вставляемая информация не совпадает по роду с той, которая находится в целевом документе, и целевой документ является составным, в документ внедряется объект.

Любая информация, которая проходит через буфер обмена, должна быть в нем зарегистрирована. Любое приложение может это сделать, а с помощью специального элемента управления можно создать объект.

Первоначальная версия OLE была реализована поверх механизма под названием DDE (*Dynamic Data Exchange*), предназначенного для асинхронной связи между приложениями. Эта версия на практике оказалась громоздкой и неуклюжей, медленно работающей или не работающей вовсе. Поэтому разработчики *Microsoft* искали способы совершенствования механизма связи и внедрения объектов.

Рождение COM

Стремясь улучшить технологию OLE, разработчики *Microsoft PowerPoint* (примерно 1993 год) 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 перевернули представление о том, как должны взаимодействовать между собой программные компоненты.

Главным понятием новой OLE стало понятие *интерфейса*.

Объект, который может предоставить сервис другим программным компонентам, делает это посредством интерфейса. Интерфейс — это список методов, действий, которые объект может выполнить. Интерфейсы объектов публикуются, описываются при помощи специальных программных структур — библиотек типов.

Вторым главным понятием является COM-объект. Новый термин COM — это *Component Object Model*, *модель компонентных объектов*.

Термины *компонентный объект* и COM-объект в данном контексте обозначают одно и то же. COM-объект представляет собой двоичную структуру (которая по сути является объектом типа C++), сформированную на основе класса, реализующего те или иные интерфейсы. Объекты COM базируются внутри программного модуля — компонента.

Механизм связи DDE, который использовался предыдущей OLE, заменили механизмом LRPC, упрощенной версией RPC. Это естественно, поскольку объекты COM, как и любые объекты в смысле объектно-ориентированного программирования, можно использовать только посредством вызовов их методов.

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

С появлением COM-объектов в программирование прочно вошло наследование интерфейсов, в противоположность наследованию классов. Интерфейсы получили широкое распространение. И это самое революционное последствие появления COM.

Взаимодействие до появления COM

Каким образом одна часть программного обеспечения получает доступ к сервисам, предоставляемой другой частью? Ответ зависит от того, что представляют собой эти части. Модель взаимодействия программного обеспечения до появления COM представлена на рисунке 2.

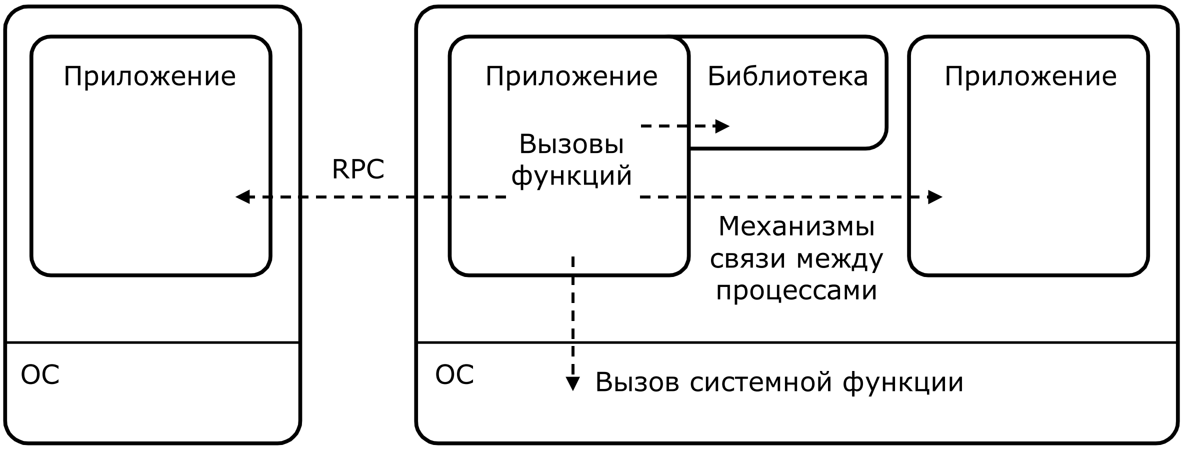


Рисунок 2 — Взаимодействие программных компонентов до COM

Приложение, скомпонованное с библиотекой, использует ее сервисы посредством прямого вызова функций. Сервисы операционной системы предоставляются через вызовы системных функций. Сервисы приложений, выполняющихся на этой или удаленной машине, можно получить через механизмы связи между процессами, — DDE, RPC, pipes.

В принципе, задача одна: как одна часть программного обеспечения получает сервисы другой части, но методы разные — системный вызов, прямой вызов функции, вызов удаленной процедуры и т.п.

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

Как работает COM

В COM любая часть программного обеспечения (программный компонент) реализует свои сервисы как один или несколько объектов COM. Каждый такой объект поддерживает один или несколько интерфейсов. Программный компонент, получающий сервис от COM-объекта, принято называть *клиентом*.

Объект COM поддерживает минимум два интерфейса, из которых как минимум один является стандартным, служебным интерфейсом, предоставляющим базовые, обязательные сервисы объекта. Стандартный интерфейс принято обозначать кружком с ручкой, направленной вверх. Неслужебные интерфейсы (*custom*-интерфейсы) предоставляют функциональность объекта, его основные сервисы.

Сам объект всегда располагается внутри некоторого сервера, которым служит либо динамически подключаемая библиотека, либо отдельный, самостоятельный процесс (приложение).

Представим себе компонент для проверки правописания в виде COM-объекта Speller. Такой объект может иметь интерфейс ISpellCheck, включающий единственный метод для проверки слова (рисунок 3).

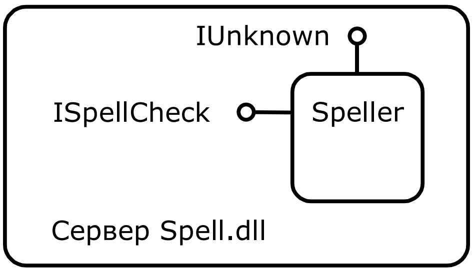


Рисунок 3 — Корректор орфографии — объект COM

Чтобы обратиться к методу интерфейса, клиент должен получить указатель на этот интерфейс при помощи библиотеки COM, являющейся частью инфраструктуры COM (рисунок 4, указатель — черная точка).

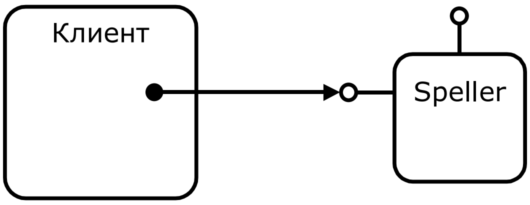


Рисунок 4 — Клиент использует указатель для доступа к интерфейсу

Библиотека COM загружает сервер в память, запрашивает у сервера объект с нужным интерфейсом, и передает интерфейс клиенту. Получив указатель на интерфейс выполняющегося объекта, клиент вызывает через этот указатель метод интерфейса.

Если позднее разработчик захочет добавить в интерфейс методы для добавления и удаления слов, то, в соответствии с правилами COM, он должен создать еще один интерфейс ISpellCheck2, который реализует метод интерфейса ISpellCheck и новые методы (рисунок 5).

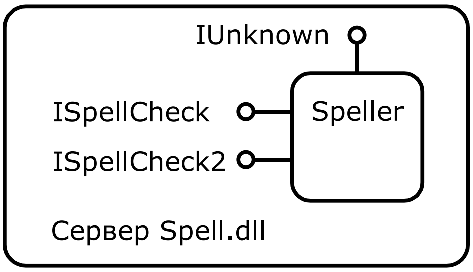


Рисунок 5 — Корректор орфографии, версия 2

После замены сервера старые клиенты объекта Speller по-прежнему будут подключаться к интерфейсу ISpellCheck, не зная о существовании нового интерфейса. Новые клиенты будут подключаться к интерфейсу ISpellCheck2, и использовать дополнительные возможности объекта.

Возможен и другой вариант модернизации объекта Speller. В этом случае новые методы формируют интерфейс ISpellModify (рисунок 6).

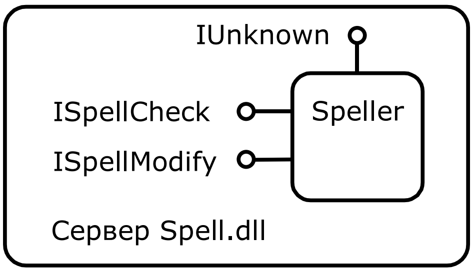


Рисунок 6 — Корректор орфографии, версия 3

Старые клиенты по-прежнему используют интерфейс ISpellCheck, а новые клиенты подключаются сразу к двум интерфейсам объекта.

Важно, чтобы выполнялся *основной закон* COM, — *если интерфейс опубликован* и начал использоваться клиентами, *изменять его нельзя*.

Модель сервисов COM

Клиенты могут получить доступ к сервисам COM-объекта только посредством интерфейса. Доступа к данным у них нет. Клиент, таким образом, полностью абстрагирован от элементов данных и устройства объекта. COM-объект, в свою очередь, полностью инкапсулирован, и представляется черным ящиком.

Вызов метода в COM соответствует вызову локальной функции. На самом деле код, выполняющийся по вызову метода, может быть частью другого потока, процесса, и вообще выполняться на другой машине. Для доступа к сервисам любых видов программного обеспечения используется одна общая модель (рисунок 7).

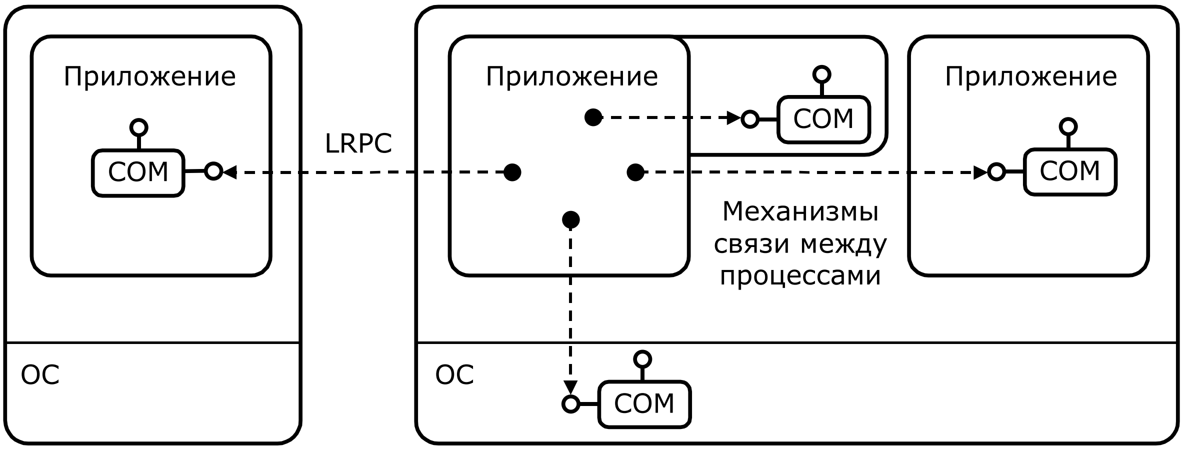


Рисунок 7 — Модель сервисов COM

COM и объектно-ориентированный подход

Системы на основе COM являются не объектно-ориентированными, а объектно-*направленными*. В COM присутствует инкапсуляция и полиморфизм, но нет ни наследования, ни иерархии. Двоичную структуру нельзя наследовать, поэтому никакой COM-объект не может являться производным от другого COM-объекта.

Инкапсуляция COM-объекта глубже, нежели инкапсуляция объекта C++, например, так как никаким образом нельзя от объекта получить то, чего нет в интерфейсе. Доступа к классу объекта нет, классы объектов являются собственностью разработчика, никак не влияющей на объекты после их публикации и распространения. Для перекомпиляции программы, использующей COM-объект, класс объекта также не требуется.

Полиморфизм COM-объектов заключается в одинаковом поведении всех COM-объектов в отношении стандартных интерфейсов, поскольку все интерфейсы COM-объект являются производными от одного из стандартных интерфейсов.

Классическое наследование реализации в COM отсутствует. Вместо этого используется *множественное наследование интерфейсов*, и это *основополагающий принцип* COM. Интерфейс есть контракт, который COM-объект обязуется выполнять. Можно создать два COM-объекта с одним и тем же интерфейсом, реализованным внутри классов объектов разными алгоритмами. Эти два объекта можно легко отличить друг от друга и использовать по очереди или одновременно (с помощью двух указателей), используя *полиморфизм* объектов.

COM и многокомпонентные программы

Многокомпонентные программы состоят из множества отдельных программных компонентов — серверов COM. В качестве примера рассмотрим приложение — игру «крестики-нолики» (рисунок 8).

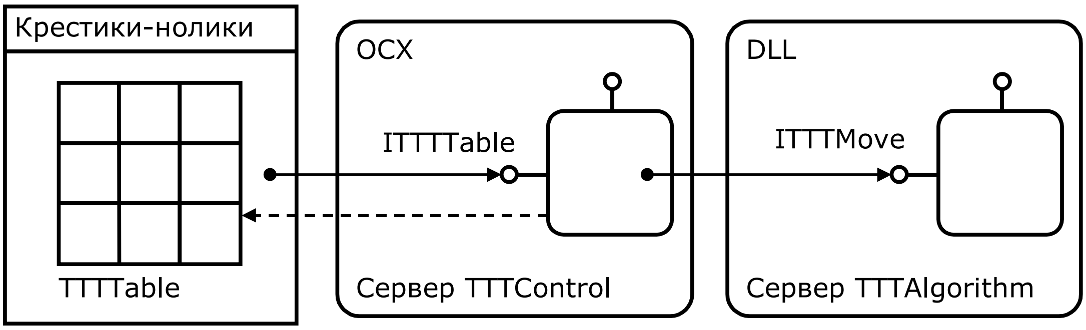


Рисунок 8 — Многокомпонентное приложение «Крестики-нолики»

В этом приложении основными элементами являются:

- интерфейс в виде игрового поля;
- алгоритм вычисления хода компьютера.

Интерфейс реализуется в виде элемента управления VBX, который предоставляется сервером элементов управления TTTControl.

Алгоритм вычисления хода компьютера реализуется в виде метода интерфейса ITTTMove, который реализуется COM-объектом сервера под названием TTTAlgorithm.

Клиентское приложение содержит окно, в котором располагается элемент управления TTTable, экспортируемый сервером TTTControl. Ход игрока вызывает метод GetBestMove() интерфейса ITTTMove. Метод вызывается элементом управления TTTable. Ответный ход компьютера также сообщается элементу управления TTTable. Таким образом, клиентское приложение состоит из одного элемента управления.

Это приложение легко модернизируется без какого-либо вмешательства в его код. Не нравится интерфейс, — установите другой компонент TTTControl. Придумали другой алгоритм, — напишите другой компонент TTTAlgorithm. Приложения, которые установили себе клиенты, не потребуют замены. Можно, например, дать возможность клиентам самим выбирать интерфейс из имеющихся в системе установленных компонентов TTTControl (при помощи дополнительных элементов управления в окне приложения).

Любой программист может добавить элемент управления в свою программу и получить дополнительное приложение к ней. Для этого нужно всего лишь подключить сервер TTTControl к своему проекту.

Этот простой пример приведен лишь для иллюстрации.

На самом деле конструкторам программ не хватает компонентов. Создание новых приложений на основе существующих протестированных компонентов имеет целью получить более надежный код быстрее и дешевле.

Объекты COM являются эффективным механизмом повторного применения программного обеспечения, так как они позволяют создавать дискретные повторно используемые бинарные компоненты.

COM решает многие проблемы. Он предоставляет стандарт на двоичный формат, в котором распространяются объекты COM. Так как 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

COM имеет практически все преимущества ООП.

COM сглаживает различие между системным и прикладным ПО.

COM не зависит от языка программирования.

COM предлагает простой подход к контролю версий.

Обзор технологий COM

Автоматизация (OLE Automation)

В любой системе обычно есть электронные таблицы, текстовые процессоры, система управления базами данных, графические редакторы, системы проектирования и т.п. В основе автоматизации лежит идея использовать возможности установленного программного обеспечения.

Для этого приложения должны быть программируемыми, то есть давать возможность программного управления ими.

Автоматизированные приложения предоставляют сервисы в виде иерархии объектов, называемой *объектной моделью приложения*.

Клиент подключается к серверу приложения, создает объекты, обращается к методам интерфейсов, получает интересующий его результат. Объектная модель позволяет перемещаться по иерархии, переходить от одного объекта к другому.

Для управления автоматизированным приложением не обязательно писать программу. Им можно управлять в буквальном смысле с помощью простейшего текстового редактора типа Блокнот.

Чтобы это было возможно, серверы автоматизации предоставляют свои сервисы через специальный интерфейс диспетчеризации `IDispatch`.

Перманентность (Persistence)

Многим объектам необходимо сохранять свои данные в течение периодов неактивности, т.е. сделать 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. Оно определяет два типа COM-объектов (рисунок 9): *storage* (хранилище, интерфейс `IStorage`) и *stream* (поток, интерфейс `IStream`).

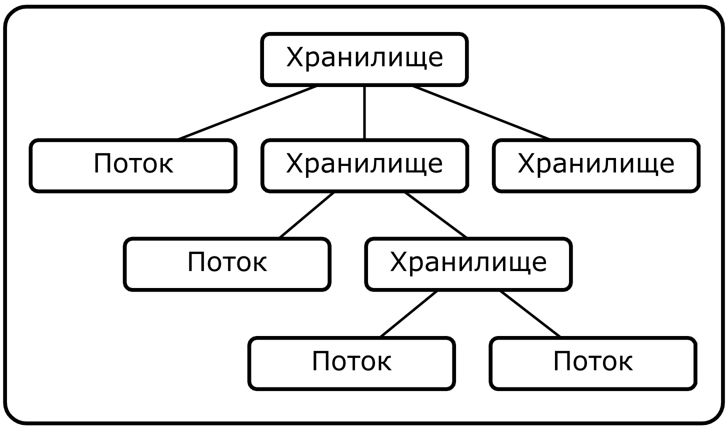


Рисунок 9 — Структурированное хранилище

Используемые для обеспечения перманентности интерфейсы имеют вид `IPersistXXX`, например, `IPersistStorage`, `IPersistFile`, `IPersistPropertyBag`.

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» жизнь программиста. Технология 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, 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.

Объекты 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 при этом используется интерфейс `IConnectionPoint`, а со стороны сервера — интерфейс `IConnectionPointContainer`. 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 предоставляет документ-контейнер. 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 составных документов используется много интерфейсов, в основном имеющих вид `IOleXXX`, — `IOleObject`, `IOleClientSite`, `IOleInPlaceObject`, `IViewObject` и другие.

Управляющие элементы ActiveX

Известны под названиями элементы VBX, OLE или OCX. Это серверы-контейнеры элементов управления, многократно используемых для создания интерфейса приложения. Элемент управления обязан наследовать только интерфейс `IUnknown`, однако на практике элементы наследуют больше интерфейсов, чем COM-объекты других типов, и обязательно интерфейсы `IOleControl`, `IOleControlSite` и некоторые другие.

OLE DB

Технология OLE DB предназначена для унифицированного доступа к базам данных различных форматов. На практике чаще используется ее упрощенная и более известная версия под названием ADO (*ActiveX Data Objects*). В основе ADO лежит объектная модель всего из 7 классов, описывающих подключения, команды, наборы данных и т.п.

Распределенная COM

Первоначальная реализация COM может работать только на одном компьютере. DCOM (*Distributed* COM) использует LRPC 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 класса.

Рассмотрим конкретный пример.

Пусть требуется создать объект, издающий звуковой сигнал.

Назовем этот объект Beeper.

Объекты Beeper издают сигнал при помощи метода Beep, а частота сигнала задается свойством Tone.

Предполагая, что в основе всей системы будет лежать наследование интерфейса, можно описать следующий класс:

```
class IBeeper {
public:
    // издает сигнал
    virtual long Beep() = 0;
    // возвращает частоту
    virtual long get_Tone(short * pVal) = 0;
    // устанавливает частоту
    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. Предполагается, что 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. Если интерфейс называется IBeeper, кокласс называется CBeeper или CoBeeper.

```
class CBeeper : public IBeeper {
public:
    // конструктор
    CBeeper() {  }
    // издает сигнал
    long Beep() { return 0; }
    // возвращает частоту
    long get_Tone(short * pVal) { return 0; }
    // устанавливает частоту
    long put_Tone(short newVal) { return 0; }
    // дополнительный метод
    void func() { printf("CBeeper::func().\n\n"); }
private:
    // данные класса и вспомогательные методы
};
```

Создавая объект кокласса, мы используем указатель на интерфейс:

```
void main() {
    IBeeper * beeper = new CBeeper();
}
```

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

```
void main() {
    IBeeper * beeper = new CBeeper();
    beeper->put_Tone(400);
    beeper->Beep();
}
```

Через указатель на интерфейс невозможно получить доступ к данным класса иначе, чем через определенные интерфейсом методы — это следствие сильной инкапсуляции объекта. Через указатель на интерфейс мы не можем вызвать метод `CBeeper::func()`.

Тем не менее, мы можем получить доступ к открытым элементам класса через переменную типа кокласса (типа `CBeeper *`) или с помощью приведения, например:

```
void main() {
    IBeeper * Beeper = new CBeeper();
    beeper->put_Tone(400);
    beeper->Beep();
    ((CBeeper*)beeper)->func();
}
```

Здесь метод `func()` будет вызван. Однако особенностью технологии является создание двоичного, а не текстового, как у нас, компонента.

Чтобы получить двоичный компонент, мы должны скомпилировать такой программный модуль, который создаст объект заданного кокласса и вернет указатель на интересующий нас интерфейс. В этом случае инкапсуляция будет полной, более того, мы не сможем узнать, как устроен кокласс, поскольку не будем располагать его текстом.

Из-за этого мы не сможем вызвать метод CBeeper::func(), используя операцию приведения или создав объект CBeeper. В этом заключается важная особенность COM — прозрачность относительного способа создания объекта, используемого языка.

В нашем распоряжении останется только некий двоичный модуль, например, DLL. Мы можем иметь несколько DLL, создающих объект, поддерживающий интерфейс IBeeper. Эти DLL могут быть написаны на разных языках. Самое главное заключается в том, чтобы внутренняя (двоичная) структура объекта во всех двоичных модулях была одной и той же — в этом случае объект можно будет использовать через указатель на интерфейс одинаковым образом в разных программных средах, в разных языках программирования.

На рисунке 10 представлена двоичная структура объекта Beeper. Она соответствует структуре классов, создаваемой *Microsoft Visual C++*.

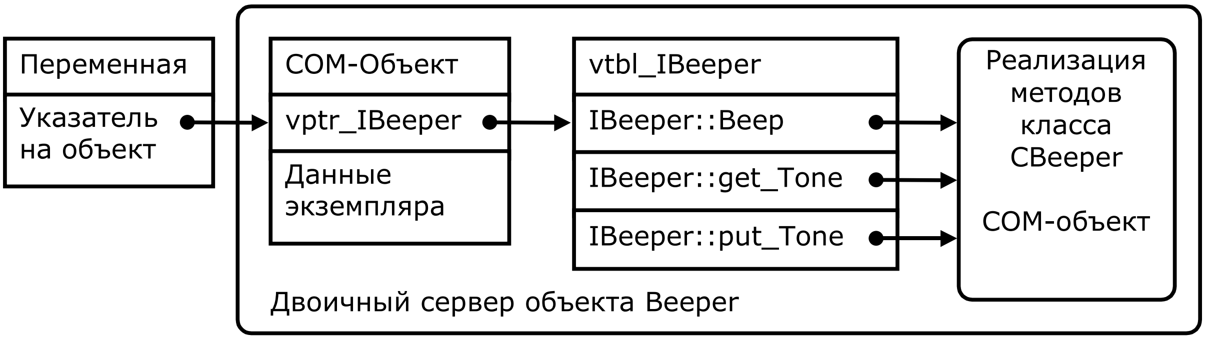


Рисунок 10 — Двоичная структура объекта Beeper

На рисунке «переменная» — это переменная beeper (указатель на интерфейс). «Объект» — это уникальный экземпляр кокласса CBeeper. Он содержит в нашем 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 для нескольких таблиц виртуальных методов. Подробнее данный аспект рассмотрен позднее, в разделе «Дуальный интерфейс», во второй части пособия.

Практическая работа AFX101

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.

Создаем приложение *Win32* на C++, платформа Win32 или x86, в зависимости от того, какая из них есть.

Выбираем тип проекта DLL.

Название проекта AFX101, размещение проекта C:\.

После создания проекта появится каталог C:\AFX101.

Откроем программу FAR Manager.

Открываем настройки F9 — Параметры — Настройки редактора.

Настройки такие:

Постоянные блоки — выключить

Автоотступ — включить

Размер табуляции — 4

OK

В FAR перейдем в каталог C:\AFX101\AFX101.

Чтобы создать текстовый файл, в FAR нужно ввести Shift+F4, после чего ввести название файла, нажать Enter, написать текст файла.

Для сохранения текста нажать Escape и Enter или F2 и Escape.

Первый файл называется AFX101.def.

Этот файл содержит описание экспортируемых функций DLL:

```
LIBRARY
EXPORTS
    DllMain PRIVATE
    DllGetBeeper PRIVATE
```

DllMain() — стандартная функция DLL, используется для инициализации DLL. DllGetBeeper() — функция, которую мы вводим в DLL для создания объекта класса CBeeper.

Убедитесь, что конец файла находится в пятой строке, а не в четвертой строке после слова PRIVATE.

Сохраните и закройте файл Escape, Enter.

Перейдем в каталог C:\AFX101. Создадим здесь два текстовых файла. Первый файл называется ibeeper.h, описывает интерфейс. Перед тем, как написать текст, выберите кодировку UTF-8 при помощи Shift+F8.

```
// ibeeper.h
#ifndef _IBEEPER_INCLUDED_
#define _IBEEPER_INCLUDED_
class IBeeper {
public:
    // издает сигнал
    virtual long Beep() = 0;
    // возвращает частоту
    virtual long get_Tone(short * pVal) = 0;
    // устанавливает частоту
    virtual long put_Tone(short newVal) = 0;
};
#endif
```

Второй файл называется cbeeper.h, описывает кокласс:

```
// cbeeper.h
#ifndef _CBEEPER_INCLUDED_
#define _CBEEPER_INCLUDED_
#include "ibeeper.h"
class CBeeper : public IBeeper {
public:
    // конструктор
    CBeeper() {
        tone = 0;
    }
    // издает сигнал
    long Beep() {
        printf("AFX101.Beeper::Beep %d.\n\n", tone);
        return 0;
    }
    // возвращает частоту
    long get_Tone(short * pVal) {
        *pVal = tone;
        return 0;
    }
    // устанавливает частоту
    long put_Tone(short newVal) {
        tone = newVal;
        return 0;
    }
    // дополнительный метод
    void func() { printf("CBeeper::func().\n\n"); }
private:
    // частота
    short tone;
};
#endif
```

Реализация методов может быть произвольной.

Возвращаемся в проект AFX101. Добавляем в проект файл AFX101.def при помощи меню Project — Add Existing Item. Так же добавляем в проект файлы интерфейса ibeeper.h и кокласса cbeeper.h.

Открываем файл stdafx.h и уточняем библиотеки:

```
// stdafx.h
#pragma once
#include "targetver.h"
//#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
```

Открываем свойства проекта AFX101.

В разделе Configuration Properties — Linker — Input установим свойство Module Definition File в значение AFX101.def. Это позволит компоновщику определить, какие функции экспортирует DLL.

Теперь перейдем в главный модуль проекта — AFX101.cpp.

Сейчас он содержит директиву включения файла stdafx.h. Описание функции DllMain() находится в файле dllmain.cpp. Здесь же мы должны описать вторую экспортируемую функцию DllGetBeeper().

```
// AFX101.cpp
#include "stdafx.h"
// описание интерфейса и кокласса
#include "..\\cbeeper.h"
// создает объект
VOID DllGetBeeper(void ** ppv) {
    // создаем объект Beeper
    IBeeper * beeper = new CBeeper();
    // возвращаем указатель на 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, создаем приложение *Win32* на C++, платформа Win32 или x86, в зависимости от того, какая из них есть. Выбираем тип проекта консольное приложение.

Название проекта TST101.

После создания каталог проекта должен появиться в C:\AFX101.

Убедитесь, что в окне Solution Explorer проект TST101 выбран.

Выберем в меню Project — Set as StartUp Project, чтобы сделать этот проект стартовым.

Открываем модуль stdafx.h нового проекта, уточняем библиотеки:

```
// stdafx.h
#pragma once
#include "targetver.h"
//#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
```

Открываем главный модуль проекта TST101.cpp и изменяем вид модуля следующим образом:

```
// TST101.cpp
#include "stdafx.h"
// описание интерфейса и кокласса
#include "..\\cbeeper.h"
// описание функции DLL
typedef VOID (*GETPROC)(void**);
// тестирование DLL
void main() {
}
```

Убеждаемся, что проект компилируется и запускается.

Теперь мы готовы к тому, чтобы провести эксперимент с коклассом и двоичным объектом, создаваемым DLL.

Сначала пробуем создать объект класса CBeeper обычным образом, так как описание класса нам сейчас доступно:

```
void main() {
    // создаем CBeeper сами
    CBeeper * cbeeper = new CBeeper();
    // вызываем методы CBeeper
    cbeeper->Beep();
    cbeeper->func();
}
```

Запускаем и убеждаемся, что методы выполняются.

Далее объявляем указатель на интерфейс и подключаем к нему объект cbeeper. Убеждаемся, что метод func() можно вызвать, используя операцию приведения:

```
void main(void) {
    // создаем CBeeper сами
    CBeeper * cbeeper = new CBeeper();
    // вызываем методы CBeeper
    cbeeper->Beep();
    cbeeper->func();
    // указатель на интерфейс
    IBeeper * beeper = 0;
    // пробуем подключить CBeeper
    beeper = cbeeper;
    // вызываем метод func
    ((CBeeper*)beeper)->func();
}
```

На самом деле при использовании двоичного компонента у нас нет текста кокласса CBeeper, как сейчас. Однако интерфейс IBeeper нам должен быть доступен в каком-то виде.

Отключаем описание кокласса:

```
// описание интерфейса и кокласса
#include "..\\ibeeper.h"
```

Теперь проект не удастся даже скомпилировать, и это понятно.

Весь приведенный выше текст в функции main() нужно теперь закомментировать, мы использовали его для демонстрации, пусть он остается в комментариях.

Главное же впереди.

Чтобы реализовать интерфейс, нам нужно загрузить DLL в память, и узнать адрес функции DllGetBeeper() при помощи системной функции GetProcAddress(). Затем при помощи указателя на функцию DllGetBeeper() мы сможем получить указатель на интерфейс.

Сначала загружаем DLL и убеждаемся, что она загрузилась:

```
void main() {
    . . .
    // загружаем DLL
    HMODULE hmod = LoadLibraryA("C:\\AFX101\\Debug\\AFX101.dll");
    if (!hmod) {
        printf("DLL load failure: %Xh.\n\n", GetLastError());
        return;
    }
}
```

Если модуль DLL загружен, получаем адрес функции DllGetBeeper():

```
void main() {
    . . .
    // указатель на функцию DLL
    GETPROC proc = (GETPROC)GetProcAddress(hmod, "DllGetBeeper");
    if (!proc) {
        printf("Function address failure: %Xh.\n\n", GetLastError());
        return;
    }
}
```

Если получить адрес функции удалось, продолжаем.

Объявляем указатель на интерфейс и пробуем его разрешить:

```
void main() {
    . . .
    // указатель на интерфейс
    IBeeper * beeper = 0;
    // вызываем функцию DLL
    (proc)((void**) & beeper);
    if (!beeper) {
        printf("DllGetObject failure: %Xh.\n\n", GetLastError());
        return;
    }
}
```

Если указатель разрешен (определен), то можно использовать его для обращения к объекту DLL, после чего выгрузить DLL из памяти:

```
void main() {
    . . .
    // вызываем свойство
    beeper->put_Tone(100);
    // вызываем метод
    beeper->Beep();
    // выгружаем DLL
    FreeLibrary(hmod);
}
```

Разработка проекта завершена, подведем итоги.

1. Идея создания объекта средствами двоичного модуля не является абсурдной, и, более того, она не является не новой.

2. Если текст класса объекта недоступен в модуле клиента, использовать можно только указатель на интерфейс, и получить при этом доступ только к методам, которые описаны в интерфейсе.

3. Поскольку непосредственно объект клиенту не доступен, нужен механизм для уничтожения объекта по окончании его использования.

4. Поскольку момент окончания использования объекта может быть неизвестен клиенту (например, когда клиент передает указатель другому клиенту), нужен механизм для определения момента выгрузки DLL.

5. Если DLL содержит объекты разных классов, нужен механизм, позволяющий выбрать класс объекта.

6. Если объект поддерживает несколько интерфейсов, нужен механизм, позволяющий выбрать интерфейс объекта.

7. Поскольку два отдельно взятых программиста могут написать два разных интерфейса с одинаковым названием, нужно иметь возможность отличить один интерфейс от другого.

8. Если для реализации двоичного объекта можно использовать различные языки, нужен какой-то механизм для описания интерфейсов, не зависящий от языка программирования.

На все затронутые вопросы мы получим ответы в последующих разделах пособия. Сейчас же заметим, что эти вопросы решаются так называемой инфраструктурой COM, состоящей из библиотеки COM, которая должна присутствовать на любой платформе, поддерживающей COM, записей в реестре, языка для описания интерфейсов, библиотек типов и других структур. 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 (файл `wtypes.h`):

```
typedef struct _GUID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} GUID;
```

Получить новый идентификатор можно при помощи приложения GUIDGEN.exe, которое входит в состав *Microsoft Visual Studio*. Сразу после запуска генерируется новый идентификатор, который можно скопировать в буфер обмена в одном из четырех форматов, а затем вставить в разрабатываемый модуль.

Программно получить новый идентификатор можно при помощи системной функции CoCreateGUID(), единственным параметром которой является структура GUID. Существуют и другие системные функции для генерации уникальных идентификаторов. Можно, например, сгенерировать множество подряд идущих идентификаторов.

На практике используются несколько разных обозначений GUID:

- 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.

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() = 0;
    virtual ULONG Release() = 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(), а также управление временем жизни созданного объекта при помощи методов AddRef() и Release().

Рассмотрим метод QueryInterface().

Параметрами метода являются:

- уникальный идентификатор интерфейса IID.
- возвращаемый указатель на запрашиваемый интерфейс.

Поскольку любой интерфейс COM наследует IUnknown, любой интерфейс имеет метод QueryInterface(). Это означает, что клиент всегда может получить указатель на любой другой интерфейс.

Управление временем жизни объекта производится при помощи специальной внутренней переменной объекта — счетчика ссылок на объект. Каждый раз, когда клиент получает ссылку на интерфейс COM-объекта, счетчик увеличивается на единицу методом AddRef(). Каждый раз, когда клиент освобождает интерфейс, он вызывает метод Release(), который уменьшает счетчик на единицу. Если счетчик достиг нулевого значения, COM-объект самоуничтожается.

Правила использования функций AddRef() и Release() просты:

- метод QueryInterface() вызывает метод AddRef() по определению; во всех других случаях (например, при копировании ссылки) клиент обязан сам вызывать метод AddRef();
- метод Release() всегда вызывается клиентом.

При программировании COM-объектов чрезвычайно важно следить за правильным вызовом указанных методов. Примеры реализации методов интерфейса IUnknown мы увидим позднее в практической работе.

HRESULT

Все методы интерфейсов должны по возможности возвращать тип HRESULT (эквивалентный unsigned long). Связано это с двоичной сущностью объектов COM.

Поскольку используемые в COM объекты сильно инкапсулированы, нет возможности проконтролировать исполнение методов интерфейсов. На объекты COM возлагается дополнительная функциональная обязанность, кроме той, которая предусматривается собственно интерфейсом, а именно, — сообщить клиенту об успешности выполнения метода.

Поэтому все типы, используемые методом для выполнения прямой функциональности, передаются объекту как аргументы, а возвращаемое значение HRESULT сообщает результат выполнения метода.

Можно сказать, что возвращаемое значение метода сообщает о результате исполнении кода, но не о результате вычисления. Результат вычисления, если он есть, в COM всегда является последним аргументом метода и помечается особым образом. Например, метод, вычисляющий синус аргумента X, должен описываться так:

```
HRESULT Sin(double X, double * Y);
```

При этом HRESULT сообщает нам, как прошло выполнение алгоритма, вычисляющего синус, а значение синуса возвращается через Y.

Заметим, что возвращаемые значения есть у методов, которые описывают функции, и методов, возвращающих значения свойств.

HRESULT — это 32-битное поле, приведенное на рисунке 11.

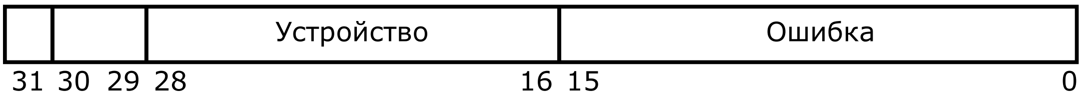


Рисунок 11 — Структура типа HRESULT

Старший бит используется для определения успешности вызова. Биты 30-29 зарезервированы. Биты 28-16 указывают на устройство, вызвавшее ошибку. Биты 15-0 указывают номер ошибки.

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 дескриптор
E_ABORT	Операция прервана
E_FAIL	Что-то не сработало

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

Пример использования макроса SUCCEEDED:

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

Нужно заметить, что если при вызове метода COM обнаружена ошибка, на ее значение указывает возвращаемое значение метода типа HRESULT. Использовать для определения ошибки функцию GetLastError() в этом случае нельзя.

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 типа 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 виртуального используется макрос PURE. Кроме того, для описания собственно интерфейса используется макрос DECLARE_INTERFACE_:

```
DECLARE_INTERFACE_(IBeeper, 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 AFX102

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

Открываем MSVS.

Создаем приложение *Win32* на C++, платформа Win32 или x86, в зависимости от того, какая из них есть.

Выбираем тип проекта DLL.

Название проекта AFX102, размещение проекта C:\.

После создания проекта появится каталог C:\AFX102.

Откроем программу FAR Manager. Находим проект AFX101.

Копируем файл AFX101.def в каталог C:\AFX102\AFX102.

Переименуем скопированный файл в AFX102.def.

Копируем файлы ibeeper.h и cbeeper.h в каталог C:\AFX102.

Перейдем в каталог C:\AFX102.

При помощи FAR создадим новый файл iid.h. Для создания файла введите Shift+F4, затем выберите кодировку UTF-8 при помощи Shift+F8.

Начальный текст файл следующий:

```
// iid.h
#ifndef _IID_INCLUDED_
#define _IID_INCLUDED_
//точка вставки
#endif
```

Запускаем программу GUIDGEN.exe, выбираем формат 2, копируем GUID в буфер обмена. Возвращаемся к файлу iid.h. В место, помеченное как *точка вставки*, вставим из буфера обмена скопированный идентификатор. В результате получим примерно следующее (идентификатор у вас будет другим):

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

Вместо **<<name>>** запишем идентификатор IID_IBeeper:

```
// iid.h
#ifndef _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
```

Этот файл содержит идентификатор и его программное название.

Переходим в проект AFX102.

Добавляем в проект все четыре файла.

Открываем свойства проекта AFX102.

В разделе Configuration Properties — Linker — Input установим свойство Module Definition File в значение AFX102.def.

Открываем файл stdafx.h и уточняем библиотеки:

```
// stdafx.h
#pragma once
#include "targetver.h"
//#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <objbase.h>
#include <initguid.h>
```

Редактируем файл ibeeper.h:

```
// ibeeper.h
#ifndef _IBEEPER_INCLUDED_
#define _IBEEPER_INCLUDED_
DECLARE_INTERFACE_(IBeeper, IUnknown) {
    // издает сигнал
    STDMETHOD(Beep)()                  PURE;
    // возвращает частоту
    STDMETHOD(get_Tone)(short * pVal)  PURE;
    // устанавливает частоту
    STDMETHOD(put_Tone)(short newVal)  PURE;
};
#endif
```

Редактируем файл cbeeper.h:

```
// cbeeper.h
#ifndef _CBEEPER_INCLUDED_
#define _CBEEPER_INCLUDED_
#include "ibeeper.h"
#include "iid.h"
class CBeeper : public IBeeper {
public:
    // конструктор
    CBeeper() {
        tone = 0;
        m_refCount = 0;
    }
    // ***** IUnknown ***** //
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv) {
        if (riid == IID_IUnknown)
            *ppv = (IUnknown*)this;
        else if (riid == IID_IBeeper)
            *ppv = (IBeeper*)this;
        else { // если интерфейса нет
            *ppv = 0;
            return E_NOINTERFACE;
        }
        ((IUnknown*)(*ppv))->AddRef();
        return S_OK;
    }
    STDMETHODIMP_(ULONG)AddRef() {
        return ++m_refCount;
    }
    STDMETHODIMP_(ULONG)Release() {
        if (--m_refCount == 0) delete this;
        return m_refCount;
    }
    // ***** IBeeper ***** //
    // издает сигнал
    STDMETHODIMP Beep() {
        printf("AFX102.Beeper::Beep %d.\n\n", tone);
        return S_OK;
    }
```

```
    // возвращает частоту
    STDMETHODIMP get_Tone(short * pVal) {
        *pVal =  tone;
        return S_OK;
    }
    // устанавливает частоту
    STDMETHODIMP put_Tone(short newVal) {
        tone = newVal;
        return S_OK;
    }
private:
    // счетчик ссылок
    ULONG m_refCount;
    // данные класса
    short tone;
};
#endif
```

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

```
// AFX102.cpp
#include "stdafx.h"
#include "..\\cbeeper.h"
// создает объект, запрашивает интерфейс
VOID DllGetBeeper(void ** ppv) {
    // создаем объект Beeper
    IBeeper * beeper = new CBeeper();
    // возвращаем указатель на интерфейс IUnknown
    beeper->QueryInterface(IID_IUnknown, (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.

Создаем приложение *Win32* на C++, платформа Win32 или x86, в зависимости от того, какая из них есть. Выбираем тип проекта консольное приложение.

Название проекта TST102.

После создания каталог проекта должен появиться в C:\AFX102.

Убедитесь, что в окне Solution Explorer проект TST102 выбран.

Выберем в меню Project — Set as StartUp Project, чтобы сделать этот проект стартовым.

Открываем модуль stdafx.h нового проекта, уточняем библиотеки:

```
// stdafx.h
#pragma once
#include "targetver.h"
//#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <objbase.h>
#include <initguid.h>
```

Открываем модуль TST102.cpp и уточняем его вид:

```
// TST102.cpp
#include "stdafx.h"
// интерфейс IBeeper
#include "..\\ibeeper.h"
// идентификатор
#include "..\\iid.h"
// описание функции DLL
typedef VOID (*GETPROC)(void**);
// основная функция
void main() {
    // указатель на IUnknown
    IUnknown * pUnk = 0;
    // указатель на IBeeper
    IBeeper * pBee = 0;

    // очистка
cleanup:
    // освобождаем указатели на интерфейсы
    if (pUnk) pUnk->Release();
    if (pBee) pBee->Release();
freelib:
    // выгружаем DLL
    FreeLibrary(hmod);
}
```

Здесь объявлены указатели на интерфейсы и описана очистка переменных и освобождение DLL.

Дальнейшие действия описываем после объявления переменных и перед очисткой.

Сначала загружаем DLL и убеждаемся, что она загружена:

```
    // загружаем DLL
    HMODULE hmod = LoadLibraryA("C:\\AFX102\\Debug\\AFX102.dll");
    if (!hmod) {
        printf("DLL load failure: %Xh.\n\n", GetLastError());
        return;
    }
    // очистка
cleanup:
```

Если DLL успешно загрузилась, доведите выполнение кода до конца, чтобы DLL выгрузилась.

Следующее действие — получить указатель на функцию DLL:

```
    // получаем указатель на функцию DLL
    GETPROC proc = (GETPROC)GetProcAddress(hmod, "DllGetBeeper");
    if (!proc) {
        printf("Function address failure: %Xh.\n\n", GetLastError());
        goto freelib;
    }
    // очистка
cleanup:
```

Если указатель на функцию получен, выполнение кода также нужно довести до конца функции main().

Далее вызываем функцию DLL и получаем указатель на IUnknown:

```
    // вызываем функцию DLL, получаем IUnknown
    (proc)((void**) & pUnk);
    if (!pUnk) {
        printf("DllGetObject failure: %Xh.\n\n", GetLastError());
        goto freelib;
    }
    // очистка
cleanup:
```

Если указатель на интерфейс получен, выполнение кода опять нужно довести до конца функции main().

Далее при помощи метода QueryInterface() получим указатель на интерфейс IBeeper:

```
    // запрашиваем IBeeper
    HRESULT hr = pUnk->QueryInterface(IID_IBeeper, (void**) & pBee);
    if (FAILED(hr)) {
        printf("DllGetObject failure: %Xh.\n\n", hr);
        goto cleanup;
    }
    // очистка
cleanup:
```

Если указатель на интерфейс IBeeper получен, выполнение кода нужно довести до конца функции main().

Остается только обратиться к свойствам и методам объекта через его интерфейс IBeeper:

```
    // вызываем свойство
    pBee->put_Tone(400);
    // вызываем метод
    pBee->Beep();
    // очистка
cleanup:
```

Если объект выполняет метод Beep(), работа успешно завершена.

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 объекты внутри этого сервера.

Фабрика класса

Объекты COM обычно находятся внутри некоторого сервера, в качестве которого выступает файл типа DLL, EXE или OCX. Создать объект, 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 требует, чтобы объекты создавались унифицированным способом.

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. «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 еще *фабриками класса* (*class factory*). Объект класса предназначен только для того, чтобы создавать объекты другого конкретного класса (кокласса).

Для каждого COM-объекта, базирующегося внутри сервера, существует своя отдельная фабрика класса. Иначе говоря, фабрика класса всегда создает объекты строго определенного класса и является, таким образом, неотъемлемой частью COM-объекта. Рисунок 12 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?

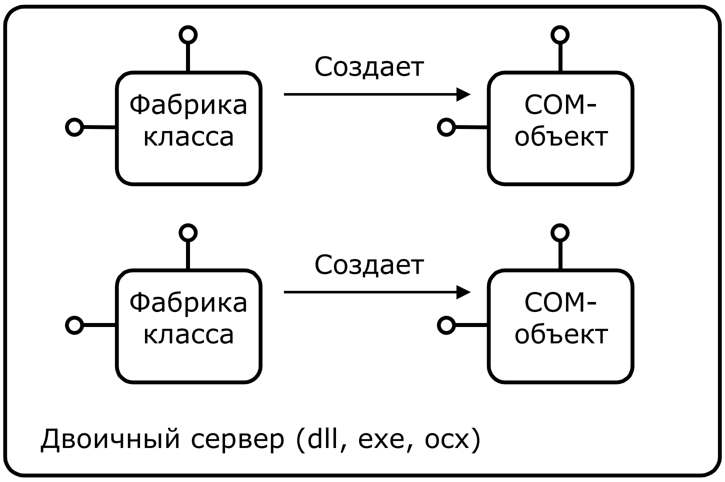


Рисунок 12 — COM-объекты и фабрики класса

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. Заметим, 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 (о реестре будет сказано немного позднее) записывается отношение двух строк:

CLSID → путь к серверу.

Зная CLSID интересующего нас кокласса, из реестра мы можем получить путь к серверу, загрузить его и получить фабрику класса.

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

1. ProgID → CLSID.
2. CLSID → ProgID.
3. CLSID → путь_к_серверу.

Первые два отношения позволяют узнать идентификатор кокласса по его *программному идентификатору* и наоборот. Третье отношение, как уже было сказано, позволяет найти сервер, зная идентификатор кокласса. Поиск идентификаторов и путей к серверам выполняет специальное программное обеспечение — библиотека COM, реализованная как набор системных функций.

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 сочетание Имя-Сервера и Имя-Кокласса, которые соединяются точкой с получением программного идентификатора ProgID.

На самом деле полный ProgID состоит из трех частей, соединяемых точками: Имя-Сервера, Имя-Кокасса и номер версии кокласса. Таким образом, ProgID = Имя-Сервера.Имя-Кокласса.Версия.

Программный идентификатор используется языками, не поддерживающими принципы классического объектно-ориентированного программирования, такими, например, как *Visual Basic* и скриптовые.

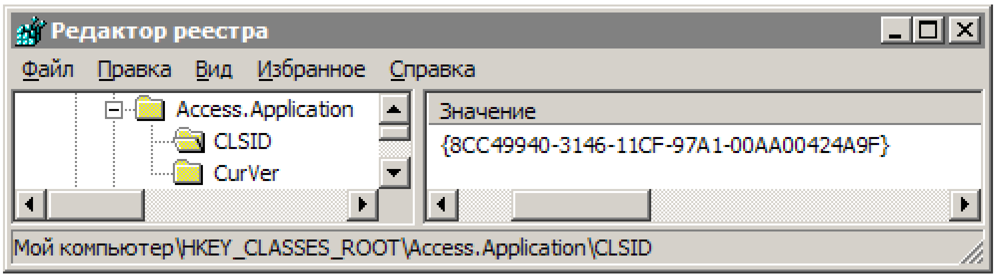
Перейдем теперь к рассмотрению собственно реестра *Windows*.

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 узлы, называемые ульями (*hives*). Точки входа под ульями называются *ключами*, они могут содержать вложенные ключи. Для разработчика COM важнейшим является улей HKEY_CLASSES_ROOT, сокращенно HKCR.

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 13 в качестве примера приведен программный идентификатор Access.Application.



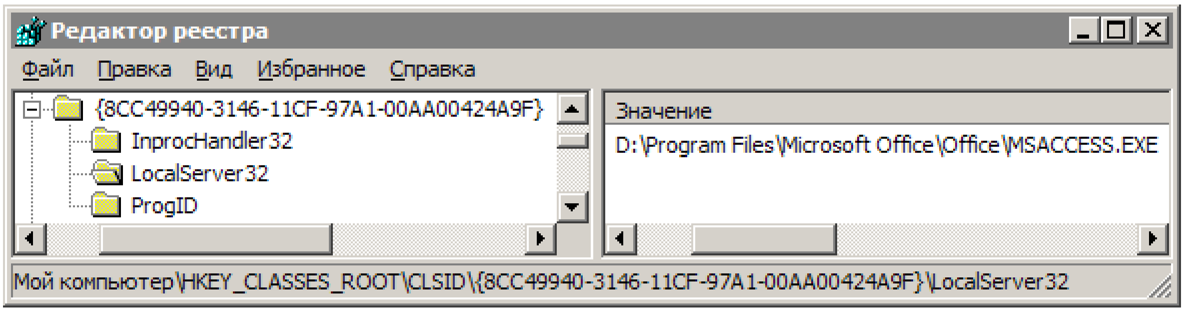
{8CC49940-3146-11CF-97A1-00AA00424A9F}

Мой компьютер\HKEY_CLASSES_ROOT\Access.Application\CLSID

Рисунок 13 — ProgID

Учитывая, что на рисунке раскрыт ключ CLSID, видно, какой CLSID соответствует этому программному идентификатору.

Далее в узле HKCR есть ключ CLSID, который содержит информацию о серверах COM. На рисунке 14 приведен ключ раздела CLSID, соответствующий ProgID Access.Application, приведенному на рисунке 13.



{8CC49940-3146-11CF-97A1-00AA00424A9F}
InprocHandler32
LocalServer32
ProgID

D:\Program Files\Microsoft Office\Office\MSACCESS.EXE

Мой компьютер\HKEY_CLASSES_ROOT\CLSID\{8CC49940-3146-11CF-97A1-00AA00424A9F}\LocalServer32

Рисунок 14 — CLSID

Учитывая, что на рисунке раскрыт ключ LocalServer32, мы можем проследить путь к соответствующему серверу в правой части редактора реестра. Ключи 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.

Нужно также знать, что информация улья HKCR формируется из узлов Software\Classes ульев HKEY_CURRENT_USER и HKEY_LOCAL_MACHINE, причем первый узел имеет преимущество перед вторым. Если в том и другом узле есть одинаковые ключи, то информация в HKCR берется из улья текущего пользователя, а не улья локальной машины.

Улей HKCR оставлен в реестре современной *Windows* для обеспечения совместимости с 16-битными версиями *Windows*.

Кроме того, в современной системе *Windows* может быть не один, а два реестра. Один из них 32-битный, а другой 64-битный. В этом случае в системе может быть три программы regedit.

В каталоге Windows находится основной редактор с именем regedit. Этот редактор отображает 64-битный реестр.

В каталоге Windows\SYSWOW64 находится 32-битная версия редактора с именем regedt32, и 64-битная версия с именем regedit. Оба этих редактора отображают 32-битный реестр.

32-битные приложения записывают и читают информацию в/из 32-битного реестра, поэтому нужно быть внимательным, выбирая редактор реестра и (или) способ регистрации.

В заключение отметим, что реестр содержит также записи об интерфейсах (ключ Interface) и библиотеках типов (ключ TypeLib).

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 regsvr32.exe.

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

```
regsvr32 C:\AFX104\BEEPC\Debug\BEEPC.dll
```

Утилита regsvr32 вызывает функцию сервера DllRegisterServer() и сообщает о результате ее выполнения при помощи стандартного диалогового окна, примерный вид которого приведен на рисунке 15.

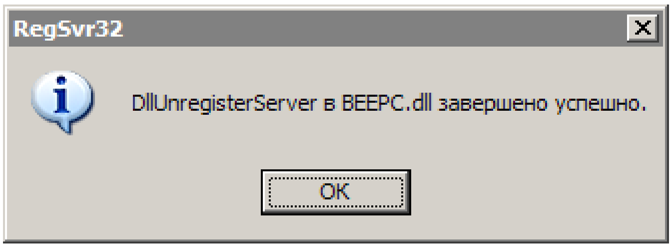
DllRegisterServer в BEEPC.dll завершено успешно.

Рисунок 15 — 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 BEEPC.dll из предыдущего примера нужно ввести командную строку:

```
regsvr32 C:\AFX104\BEEPC\Debug\BEEPC /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, примерный вид которого приведен на рисунке 16.



DllUnregisterServer в BEEPC.dll завершено успешно.

Рисунок 16 — 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`.

Автоматическая регистрация сервера производится при его создании (компиляции) средствами *Microsoft Visual Studio*.

Зарегистрировать сервер типа DLL можно также при помощи файла с расширением `.reg`. Первая строка файла должна содержать `REGEDIT4`, последующие строки — записываемые отношения.

Например, для регистрации сервера `AFX103.dll`, который будет разработан в очередной практической работе, можно создать следующий текстовый файл с именем `AFX103.reg` (рисунок 17).

```
REGEDIT4

[HKEY_CLASSES_ROOT\AFX103.Beeper]

[HKEY_CLASSES_ROOT\AFX103.Beeper\CLSID]
@="{...}"

[HKEY_CLASSES_ROOT\CLSID\{...}]
@="AFX100 Beeper Class"

[HKEY_CLASSES_ROOT\CLSID\{...}\InprocServer32]
@="C:\\AFX103\\Debug\\AFX103.dll"

[HKEY_CLASSES_ROOT\CLSID\{...}\ProgID]
@="AFX103.Beeper"
```

Рисунок 17 — Файл AFX103.reg

Вместо "`{...}`" записывается идентификатор кокласса, например:
`{3C66CCE7-95A1-4F47-A8F3-2B1A975C8A2E}` .

Для регистрации файл типа `.reg` можно «запустить».

В случае, если требуется записать информацию в 32-битный реестр в системе, где есть 64-битный реестр, нужно открыть редактор, отображающий 32-битный реестр, и выполнить *импорт* файла `.reg`.

Еще лучше в этом случае выполнить команду:

```
C:\Windows\SysWOW64\reg  import  C:\AFX103\AFX103.reg
```

Здесь `C:\Windows` — каталог системы, который может быть другим, например, `C:\WINNT`; `C:\AFX103\AFX103.reg` — путь к файлу `.reg`; модификатор `import` указывает на операцию. Команду можно ввести в командной строке файлового менеджера FAR, или в окне Выполнить.

Библиотека COM

На вычислительной системе, поддерживающей COM, должна присутствовать *библиотека* COM, состоящая из набора функций `OLE32.dll`. Каждая функция библиотеки COM начинается с префикса `Co`.

В этом разделе рассматриваются только некоторые из функций.

`CoInitialize()` — инициализирует библиотеку COM для текущего потока. Параметр всегда равен `0`. Эта функция должна вызываться первой. При этом текущий поток входит в однопоточный апартамент. Для входа в многопоточный апартамент используют функцию `CoInitializeEx`.

`CoUninitialize()` — закрывает библиотеку COM для текущего потока, и выводит поток из текущего апартамента.

Для создания объекта используют функции, которые управляются менеджером сервисов SCM (*Service Control Manager*, `RPCSS.exe`, причем это не тот менеджер SCM, который управляет сервисами NT).

`CoCreateInstance()` — создает одиночный объект COM на локальной машине. Для создания одиночного объекта COM на удаленной машине используют функцию `CoCreateInstanceEx()`.

Параметры функции:

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`.

Обычно используются две константы:

`CLSCTX_INPROC_SERVER` для создания объекта в процессе клиента;

`CLSCTX_LOCAL_SERVER` для создания объекта вне процесса клиента.

`CoGetClassObject()` — создает фабрику класса заданного `CLSID` и возвращает указатель на запрашиваемый интерфейс. Используется при создании множества объектов.

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 возвращают отношения из реестра.

`CLSIDFromProgID(LPOLESTR, LPCLSID)` — возвращает `CLSID` по известному `ProgID`. Параметры функции: `ProgID` и `CLSID`.

`ProgIDFromCLSID(REFCLSID, LPOLESTR)` — возвращает `ProgID` по известному `CLSID`. Параметры функции: `CLSID` и `ProgID`.

Практическая работа AFX103

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 (текстов) из предыдущей работы AFX102.

Открываем MSVS.

Создаем приложение *Win32* на C++, платформа Win32 или x86, в зависимости от того, какая из них есть.

Выбираем тип проекта DLL.

Название проекта AFX103, размещение проекта C:\.

После создания проекта появится каталог C:\AFX103.

Откроем программу FAR Manager. Находим проект AFX102.

Копируем файл AFX102.def в каталог C:\AFX103\AFX103.

Переименуем скопированный файл в AFX103.def.

Копируем файлы iid.h, ibeeper.h и cbeeper.h в каталог C:\AFX103.

Переходим в проект AFX103.

Добавляем все четыре файла в проект.

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

```
// stdafx.h
#pragma once
#include "targetver.h"
//#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <objbase.h>
#include <initguid.h>
```

Открываем модуль AFX103.def.

Редактируем его, нам нужно экспортировать две из функций DLL:

```
LIBRARY
EXPORTS
    DllMain PRIVATE
    DllCanUnloadNow PRIVATE
    DllGetClassObject PRIVATE
```

В свойствах проекта устанавливаем Linker — Input = AFX103.def.

Открываем модуль iid.h.

В него нужно добавить идентификатор кокласса CLSID_Beeper.

Для этого сгенерируйте новый GUID при помощи GUIDGEN.exe, выберите форму 2, скопируйте идентификатор в буфер обмена.

Далее перейдите в проект AFX103 и вставьте идентификатор, так, чтобы получить следующий примерный вид модуля iid.h:

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

Перейдем в модуль AFX103.cpp.

Объявим в нем счетчики модуля, учитывающие количество блокировок сервера и количество активных объектов:

```
// AFX103.cpp
#include "stdafx.h"
// счетчик объектов
ULONG g_objCount = 0;
// счетчик блокировок
ULONG g_lockCount = 0;
```

Для описания фабрики класса добавим в проект заголовочный файл с названием factory.h (при помощи меню Project — Add New Item).

Примерный код этого файла приведен ниже:

```
// factory.h
#ifndef _FACTORY_H_INCLUDED_
#define _FACTORY_H_INCLUDED_
#include <windows.h>
extern ULONG g_objCount;
extern ULONG g_lockCount;
// фабрика класса
class BeeperFactory : public IClassFactory {
public:
    BeeperFactory() {
        m_refCount = 0;
        ++g_objCount;
    }
    ~BeeperFactory() {
        --g_objCount;
    }
    /***** IUnknown *****/
    STDMETHODIMP 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;
    }
```

```
    STDMETHODIMP_(ULONG)AddRef() {
        return ++m_refCount;
    }
    STDMETHODIMP_(ULONG)Release() {
        if (--m_refCount == 0) delete this;
        return m_refCount;
    }
    /***** IClassFactory *****/
    STDMETHODIMP CreateInstance(LPUNKNOWN, REFIID riid, void** ppv) {
        CBeeper * pBeeper = 0;
        // создаем объект Beeper
        pBeeper = new CBeeper;
        // запрашиваем интерфейс
        return pBeeper->QueryInterface(riid, ppv);
    }
    STDMETHODIMP LockServer(BOOL fLock) {
        if (fLock) ++g_lockCount; else --g_lockCount;
        return S_OK;
    }
private:
    ULONG m_refCount;
};
#endif
```

Перейдем к реализации функций сервера.

Модуль AFX103.cpp.

В начале модуля включаем в него необходимые модули:

```
// AFX103.cpp
#include "stdafx.h"
#include "..\\iid.h"
#include "..\\cbeeper.h"
#include "factory.h"
// счетчик объектов
ULONG g_objCount = 0;
// счетчик блокировок
ULONG g_lockCount = 0;
```

Далее описываем две стандартные функции DLL:

```
// можно выгрузить эту DLL?
STDAPI DllCanUnloadNow() {
    if (g_lockCount == 0 && g_objCount == 0) return S_OK;
    return S_FALSE;
}

// возвращает интерфейс фабрики класса
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, void ** ppv) {
    // этот сервер создает только Beeper
    if (rclsid != CLSID_Beeper) return CLASS_E_CLASSNOTAVAILABLE;
    // фабрика класса Beeper
    BeeperFactory * factory = new BeeperFactory;
    // получаем и возвращаем интерфейс
    return factory->QueryInterface(riid, ppv);
}
```

Построим проект и устраним ошибки, если они есть.

Итак, настоящий сервер COM готов.

Для его тестирования добавим новый проект.

Выбираем в меню File — Add — New Project.

Создаем приложение *Win32* на C++, платформа Win32 или x86, в зависимости от того, какая из них есть.

Выбираем тип проекта консольное приложение.

Название проекта TST103.

После создания каталог проекта должен появиться в C:\AFX103.

Убедитесь, что в окне Solution Explorer проект TST103 выбран.

Выберем в меню Project — Set as StartUp Project, чтобы сделать этот проект стартовым.

Открываем модуль stdafx.h нового проекта, уточняем библиотеки:

```
// stdafx.h
#pragma once
#include "targetver.h"
//#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <objbase.h>
#include <initguid.h>
```

Открываем модуль TST103.cpp и уточняем его вид:

```
// TST103.cpp
#include "stdafx.h"
// интерфейс IBeeper
#include "..\\ibeeper.h"
// идентификатор
#include "..\\iid.h"

// CoGetClassObject
void main_a() {
}

// CoCreateInstance
void main_b() {
}

// основная функция
void main() {
    CoInitialize(0);
    main_a();
    main_b();
    CoUninitialize();
}
```

Мы будем получать указатель на интерфейс IBeeper двумя разными путями, — при помощи функции CoGetClassObject() и при помощи функции CoCreateInstance().

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

```
void main_a() {
    // указатели на интерфейсы
    IClassFactory * pFact = 0;
    IBeeper * pBeep = 0;
    // указатель на фабрику класса
    HRESULT hr = CoGetClassObject(CLSID_Beeper,
        CLSCTX_INPROC_SERVER, 0, IID_IClassFactory, (void**) & pFact);
    if (FAILED(hr)) {
        printf("CoGetClassObject failure: %Xh\n\n.", hr);
        goto cleanup;
    }
    hr = pFact->CreateInstance(0, IID_IBeeper, (void**) & pBeep);
    if (FAILED(hr)) {
        printf("CreateInstance failure: %Xh\n\n.", hr);
        goto cleanup;
    }
    // вызываем свойство
    pBeep->put_Tone(400);
    // вызываем метод
    pBeep->Beep();
cleanup:
    if (pFact) pFact->Release();
    if (pBeep) pBeep->Release();
}
```

Компилируем и запускаем проект на выполнение.

Убеждаемся, что создать интерфейс IClassFactory невозможно. Дело в том, что сервер не зарегистрирован в реестре операционной системы.

Сейчас в каталоге AFX103 нужно создать текстовый файл AFX103.reg и записать в нем текст, приведенный на рисунке 17. При этом нужно заменить "…" кодом идентификатора CLSID_Beeper

Далее нужно попытаться «выполнить» файл AFX103.reg. В FAR для этого нужно установить указатель на файл и нажать Enter.

Если регистрация завершится успешно, нужно попробовать заново выполнить код функции main_a().

Если код выполнится успешно, то регистрация прошла правильно, можно продолжать работу, описывать функцию main_b().

Если код функции main_a() по-прежнему не выполняется успешно, следует выполнить следующую команду, например, в FAR:

```
C:\Windows\SysWOW64\reg  import  C:\AFX103\AFX103.reg
```

Если и в этом случае код функции main_a() не приводит к успеху, нужно обратиться к преподавателю.

В этом месте нужно быть уверенным, что *сервер должным образом зарегистрирован в реестре*, после чего можно описать функцию main_b.

В функции main_b() мы пойдем более коротким путем:

```
void main_b() {
    // указатель на интерфейс
    IBeeper * pBeep = 0;
    // сразу получаем указатель
    HRESULT hr = CoCreateInstance(CLSID_Beeper, 0,
        CLSCTX_INPROC_SERVER, IID_IBeeper, (void**) & pBeep);
    if (FAILED(hr)) {
        printf("CoCreateInstance failure: %Xh\n\n.", hr);
        goto cleanup;
    }
    // вызываем свойство
    pBeep->put_Tone(800);
    // вызываем метод
    pBeep->Beep();
cleanup:
    if (pBeep) pBeep->Release();
}
```

Убеждаемся, что объект создается и выполняет метод.

Таким образом, мы создали сервер COM-объекта, успешно взаимодействующий с инфраструктурой COM. Однако есть еще вопросы, которые должны быть решены в рамках COM. Это языково-независимое описание и публикация интерфейсов, взаимодействие кода клиента и кода объекта, исполняемых в разных процессах, 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 сервера типа DLL или OCX, загружаемого 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 сервера 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 сервера любого типа. В реестре эти серверы не записываются — они являются серверами *in-process* или локальными серверами на другой машине, и записываются в реестре ее операционной системы. Удаленные серверы используют DCOM.

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*, *транспортировка*).

Примечание: вообще говоря, маршалинг выполняется не только для объекта в другом процессе, но и для объекта в другом потоке текущего процесса, если указатель передается через границу так называемого *апартамента* (апартаменты см. далее).

Общая схема реализации передачи указателя на интерфейс объекта, расположенного в другом процессе, приведена на рисунке 18.

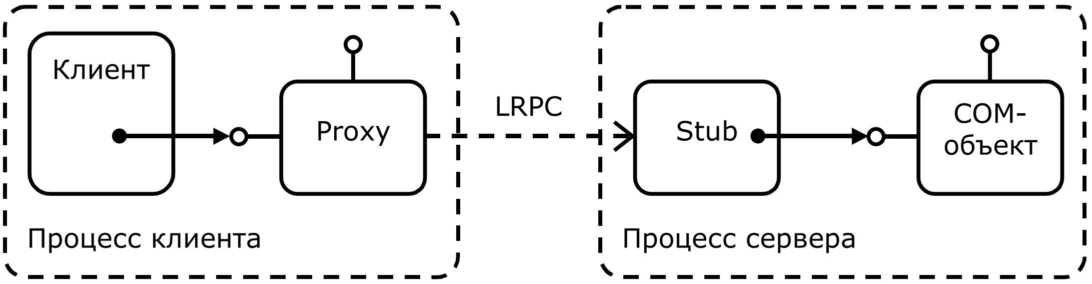


Рисунок 18 — Маршалинг интерфейса

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 (потока, апартамента). Заместитель является COM-объектом, реализующим те же интерфейсы, что и COM-объект, находящийся в другом процессе (потоке, апартаменте).

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 или RPC 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 соответствует типам языка *Visual Basic 6.0*, а 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 предоставить собственную реализацию интерфейса `IMarshal`.

Апартаменты

Потоковая модель COM, учитывающая разные потоковые модели клиентов, основана на абстрактном понятии *апартамента* (*apartment*).

Все COM-объекты процесса поделены на группы с одинаковыми требованиями к многопоточности, эти группы называют апартаментами.

Существует два типа апартаментов.

1. Однопоточный апартамент STA (*single-threaded apartment*).
2. Многопоточный апартамент MTA (*multithreaded apartment*).

Однопоточный апартамент

В STA может выполняться только один поток.

Апартамент типа STA упорядочивает обращения к своим объектам.

При этом COM использует очередь сообщений *Windows*, связанную со скрытым окном, создаваемым каждым STA. Вызовы методов следуют один за другим в порядке очередности, ни один вызов не может начаться прежде, чем закончится предыдущий вызов.

Программист не должен заботиться о синхронизации или реентерабельности, и не обязан обеспечивать защиту данных объекта от последствий параллельного доступа. Данный апартамент значительно облегчает программирование, но выполнение методов менее эффективно.

Многопоточный апартамент

В MTA могут выполняться параллельно несколько потоков.

Апартамент типа MTA не выполняет никакого упорядочивания обращений к своим объектам. Потоки, выполняющиеся в MTA, могут обращаться к методам одного и того же объекта параллельно. Выполняющиеся MTA потоки иногда называют *свободными* (*free*).

Поскольку упорядочивания вызовов методов нет, код методов должен обеспечивать синхронизацию и реентерабельность самостоятельно, применяя примитивы синхронизации. Этот апартамент усложняет программирование объекта, но дает выигрыш в производительности за счет параллельности и реентерабельности.

Процессы, потоки и апартаменты

Процесс может иметь несколько STA и ровно один MTA.

Апартамент типа STA создается, когда поток входит в него, вызывая `CoInitialize`. Первый созданный STA называют главным (*main*).

Апартамент типа MTA создается при первом вызове `CoInitializeEx`, в случае, если параметр указывает на MTA.

Поток, вошедший в апартамент, не может его менять до выхода из апартамента. Выход из апартамента выполняет `CoUninitialize`.

COM-классы и апартаменты

Каждый `CLSID` имеет свою потоковую модель, которая записывается в реестре под ключом `InprocServer32\ThreadingModel`.

Кокласс может иметь одну из четырех потоковых моделей.

1. `Single`. Это модель взаимодействия клиента и объекта до появления апартаментов. Объект выполняется в главном STA, который является единственным. Соответствует отсутствию ключа `ThreadingModel`.
2. `Apartment`. Объекты могут выполняться во многих потоках, но при этом каждый объект находится в своем STA. Это дает возможность использовать объекты в многопоточном приложении, однако вызовы синхронизируются и параллельность исключается.
3. `Free`. Все объекты выполняются в MTA. Маршалинг в пределах MTA не выполняется, это дает наивысшую производительность.
4. `Both`. Объект может выполняться в любом апартаменте.

COM-объекты и апартаменты

Объект COM всегда существует ровно в одном апартаменте.

При создании объекта он помещается в апартамент, зависящий от его модели. Если поток клиента находится в STA, объект находится в этом STA. Если поток клиента находится в MTA, объект находится либо в этом MTA, либо в *главном* STA, который создает COM, если модель потоков класса объекта не многопоточная.

Межпоточный маршалинг

Чтобы получить указатель на интерфейс из другого потока данного процесса, требуется явно выполнять маршалинг при помощи функций CoMarshalInterThreadInterfaceInStream или CoGetInterfaceAndReleaseStream.

IDL

Для описания интерфейсов, классов и других объектов используется язык IDL (*Interface Definition Language*), произошедший, в свою очередь от языка ODL (*Object Definition Language*). Эти языки имеют синтаксис, схожий с синтаксисом Си. Файлы имеют расширение .idl или .odl, и компилируются при помощи, например, MIDL (*Microsoft* 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("Beeps")] 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, также 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 следующая 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.

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 19).

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

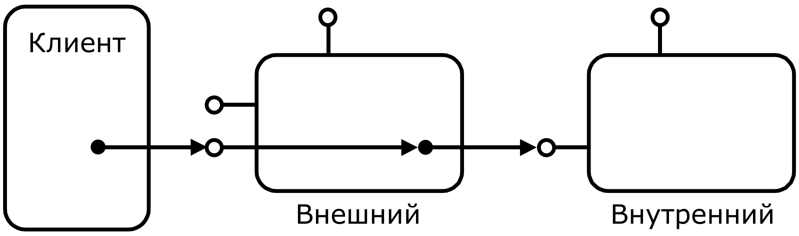


Рисунок 19 — Делегирование интерфейса

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.

2. Агрегация (*Aggregation*). Включение замедляет процесс вызова метода внутреннего объекта. При агрегировании внешний объект выставляет интерфейсы внутреннего объекта как свои собственные (рисунок 20). Клиент, обращаясь к интерфейсу внешнего объекта, на самом деле получает указатель на интерфейс внутреннего объекта.

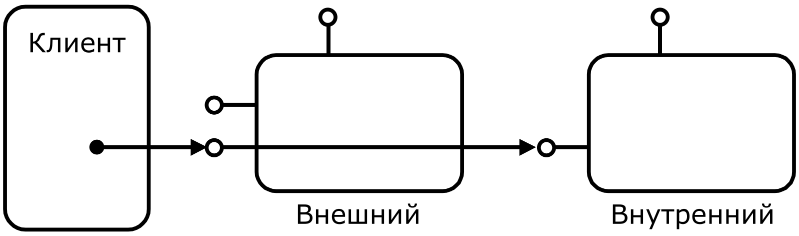


Рисунок 20 — Агрегирование интерфейса COM-объекта

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 AFX104

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

Создание сервера

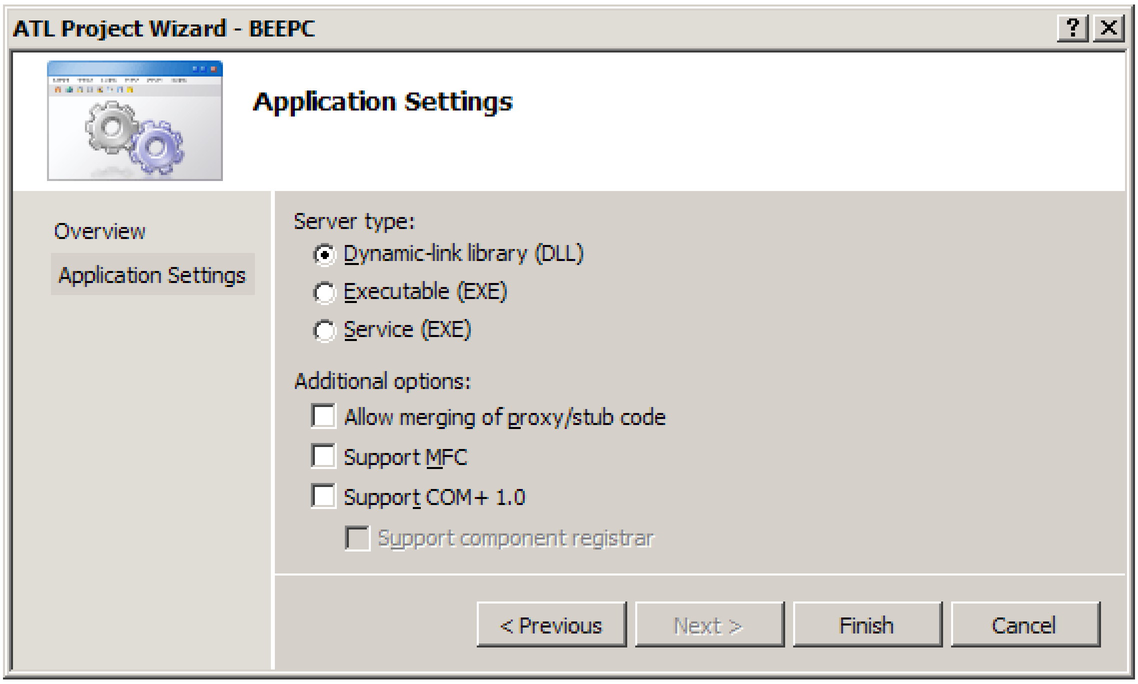
Создаем новый проект C++, раздел *ATL*, шаблон *ATL Project*.

В мастере *ATL Project Wizard* вводим:

- имя проекта `BEEPC` ,
- расположение проекта `C:\` ,
- имя решения (*solution*) `AFX104` .

Заметим, что название проекта становится именем сервера, которое будет регистрироваться в реестре, поэтому к выбору названия проекта нужно подходить обдуманно.

Первый шаг мастера ничего не содержит, переходим ко второму. На рисунке 21 показан примерный вид второго шага.



Server type:

● Dynamic-link library (DLL)
○ Executable (EXE)
○ Service (EXE)

Additional options:

☐ Allow merging of proxy/stub code
☐ Support MFC
☐ Support COM+ 1.0
☐ Support component registrar

Рисунок 21 — 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* включает код маршалинга непосредственно в проект. Флажок *Support MFC* управляет поддержкой MFC (*Microsoft Foundation Classes*). Мы оставляем эти флажки выключенными. Нажимаем кнопку *Finish*.

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

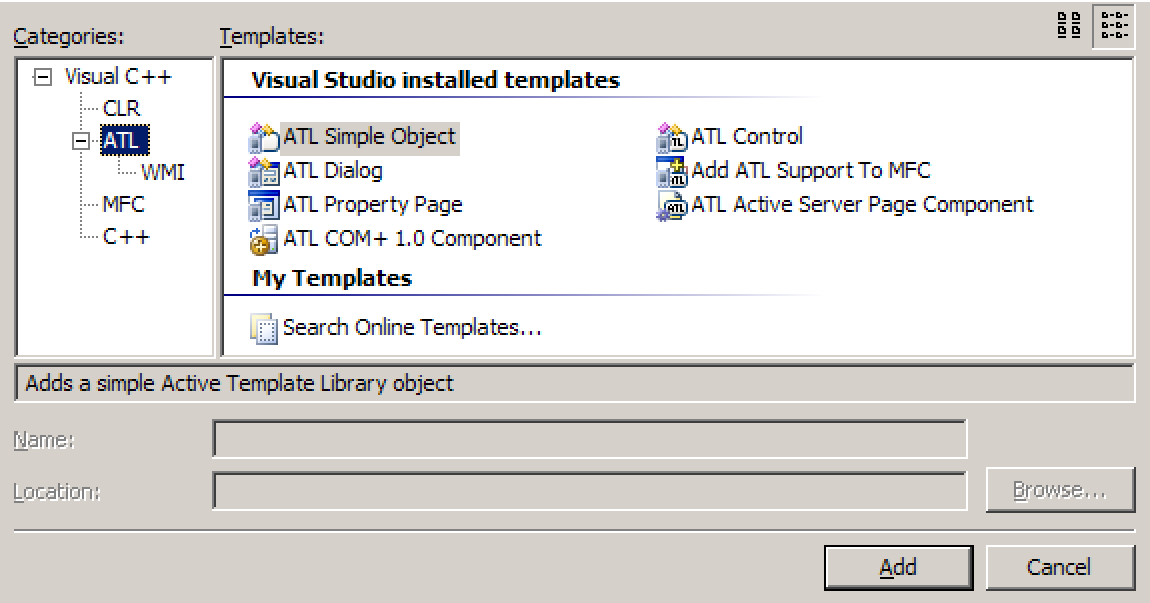
Второй файл, который мы рассмотрим, — основной файл проекта BEEPC.cpp. Он содержит реализацию функций DLL. Заметим, что ATL берет на себя всю рутинную работу по созданию сервера. Современные версии MSVS включают в состав функций DLL еще одну, с названием DllInstall(). Она позволяет регистрировать сервер в необходимой части реестра (HKLM или HKCU):

```
// DllInstall - Adds/Removes entries to the system registry per user
//              per machine.
STDAPI DllInstall(BOOL bInstall, LPCWSTR pszCmdLine)
{
    . . .
}
```

Кроме того, решение 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 22).

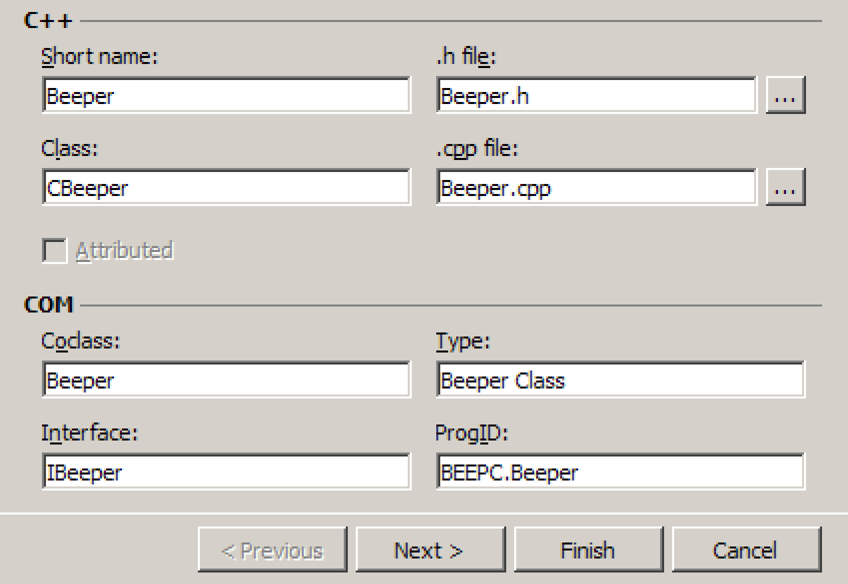


Adds a simple Active Template Library object

Рисунок 22 — 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 23).



Short name: Beeper

.h file: Beeper.h

Class: CBeeper

.cpp file: Beeper.cpp

☐ Attributed

Coclass: Beeper

Type: Beeper Class

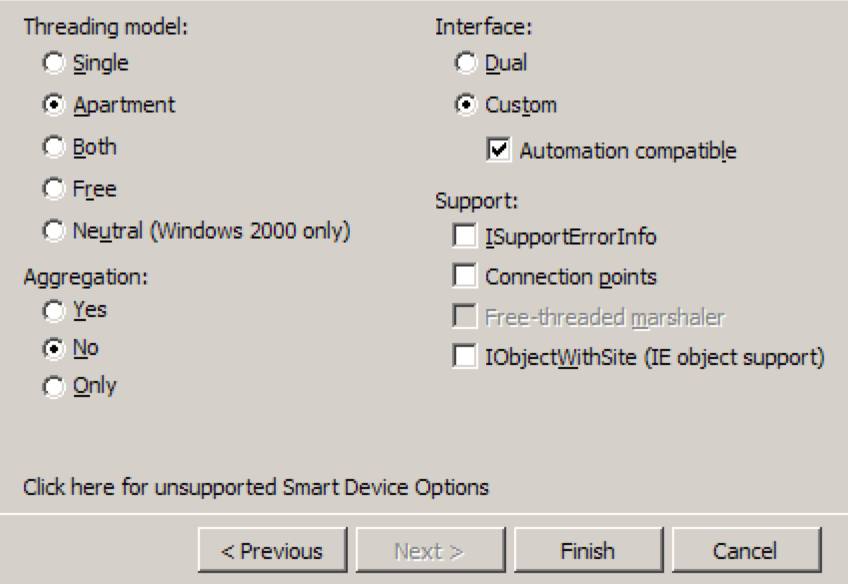
Interface: IBeeper

ProgID: BEEPC.Beeper

Рисунок 23 — Определение имен COM-объекта

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

На следующем шаге (рисунок 24) выбирается потоковая модель, базовый интерфейс и агрегация объекта.



Threading model:
○ Single
● Apartment
○ Both
○ Free
○ Neutral (Windows 2000 only)

Aggregation:
○ Yes
● No
○ Only

Interface:
○ Dual
● Custom
☑ Automation compatible

Support:
☐ ISupportErrorInfo
☐ Connection points
☐ Free-threaded marshaler
☐ IObjectWithSite (IE object support)

Click here for unsupported Smart Device Options

Рисунок 24 — Свойства кокласса

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*).

Агрегирования объектов у нас не будет, поэтому выбираем переключатель *NO*.

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:

```
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 макроопределение:

```
DECLARE_REGISTRY_RESOURCEID(IDR_BEEPER)
```

Оно определяет идентификатор ресурса, в котором хранится описание класса. Далее следует макроопределение `COM_MAP`, которое описывает экспортируемые интерфейсы. Каждый интерфейс описывается при помощи макроопределения `COM_INTERFACE_ENTRY`. Клиент может получить указатель только на тот интерфейс, который есть в данной таблице:

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

Следующее макроопределение предназначено для защиты объекта от уничтожения, если во время выполнения функции FinalConstruct() будет неудачно создан агрегированный объект:

```
DECLARE_PROTECT_FINAL_CONSTRUCT()
```

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 описывает информацию для регистрации в реестре.

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 25).

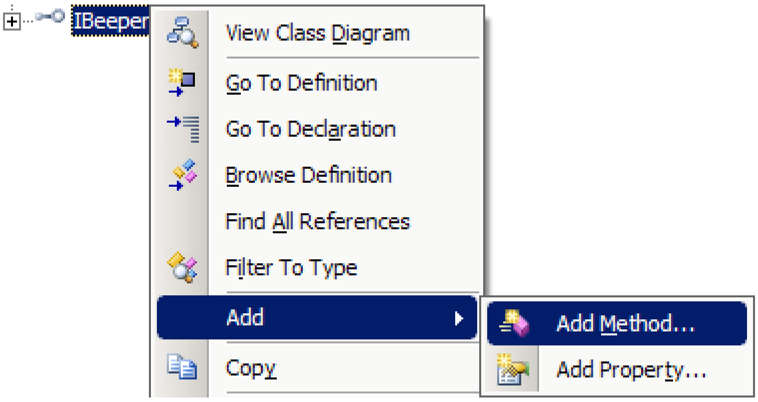
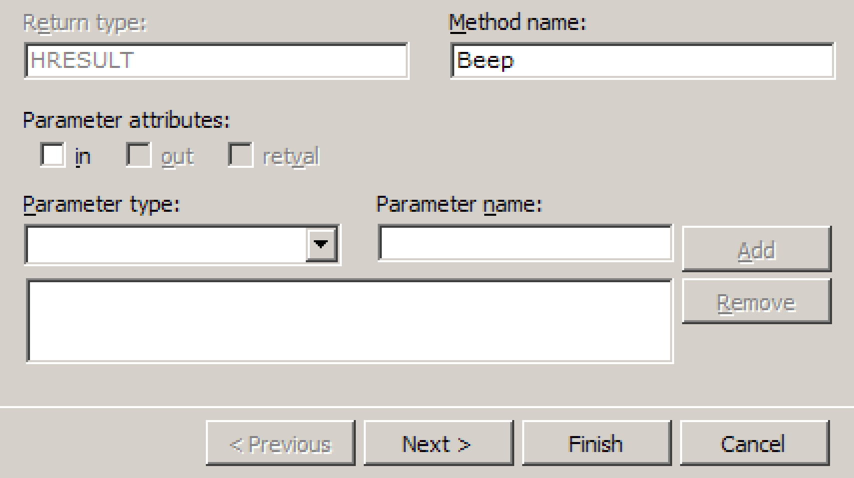


Рисунок 25 — Добавление в интерфейс метода или свойства

Далее нужно найти интерфейс `IBeeper` и выбрать в контекстном меню `Add — Add Method`. Найти нужный пункт `IBeeper` может оказаться не таким простым делом. Нужно попробовать несколько раз.

В результате должен появиться мастер *Add Method Wizard*, примерный вид которого показан на рисунке 26.

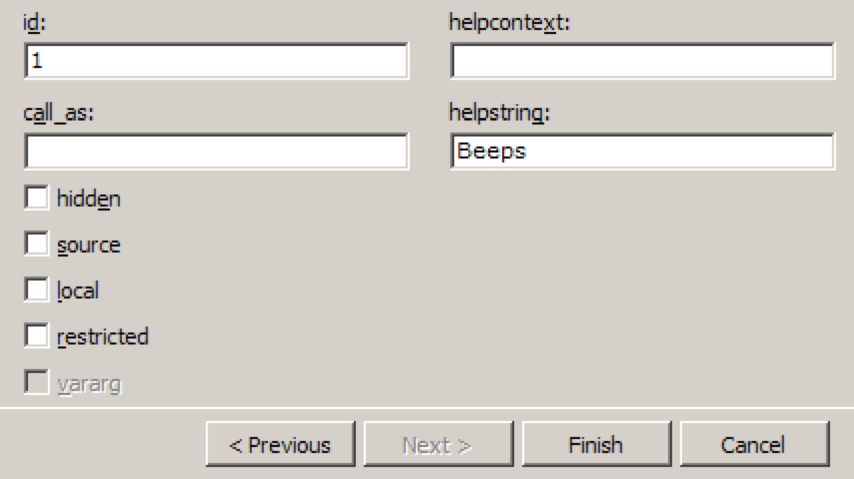


Return type: HRESULT
Method name: Beep
Parameter attributes: ☐ in ☐ out ☐ retval
Parameter type:
Parameter name:

Рисунок 26 — 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 27).



id: 1
helpcontext:
call_as:
helpstring: Beeps
☐ hidden
☐ source
☐ local
☐ restricted
☐ vararg

Рисунок 27 — 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("Beeps")] HRESULT Beep();
    };
```

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

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

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

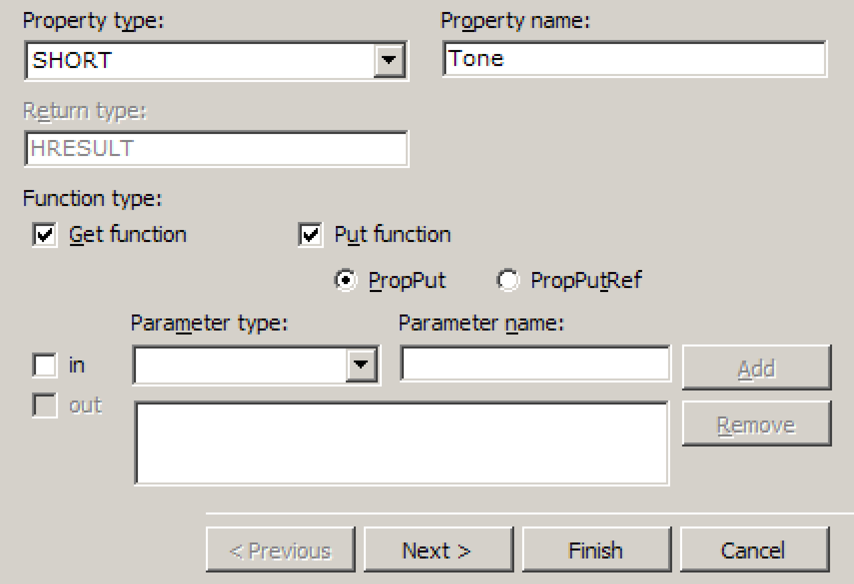
```
STDMETHODIMP CBeeper::Beep() {
    // TODO: Add your implementation code here
    return S_OK;
}
```

Добавление свойства

Добавление свойства выполняется аналогично добавлению метода.

Открываем контекстное меню интерфейса IBeeper (рисунок 25) и выбираем в нем Add — Add Property.

Появится диалог для добавления свойства (рисунок 28).



Property type: SHORT

Property name: Tone

Return type: HRESULT

Function type:

☑ Get function ☑ Put function

● PropPut ○ PropPutRef

Parameter type:

Parameter name:

☐ in

☐ out

Рисунок 28 — 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 создать свойство только для чтения или свойство 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:

```
interface IBeeper : IUnknown {
    [id(1), helpstring("Beeps")] 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.cpp появились функции свойства (см. ниже).

В файле Beeper.h также появились описания функций свойства:

```
public:
    STDMETHOD(Beep)(void);
    STDMETHOD(get_Tone)(SHORT* pVal);
    STDMETHOD(put_Tone)(SHORT newVal);
};
```

Добавляем секцию private и описываем элемент данных:

```
    STDMETHOD(put_Tone)(SHORT newVal);
private:
    SHORT m_tone;
};
```

В конструкторе кокласса устанавливаем начальное значение:

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

В модуле Beeper.cpp нужно подключить библиотеку <stdio.h>, затем описать функции кокласса:

```
STDMETHODIMP CBeeper::Beep() {
    printf("BEEPC::Beep: %d.\n\n", m_tone);
    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.

Тестовый проект на VB

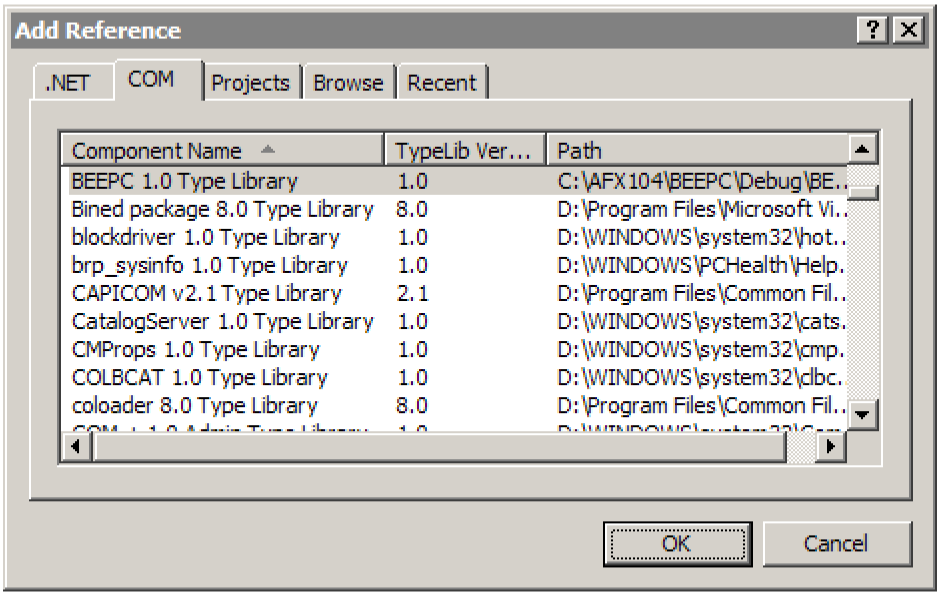
Для тестирования сервера добавим проект на языке *Visual Basic*.

Выбираем в меню `File — Add — New Project`. Язык *Visual Basic*, раздел *Windows*, шаблон *Console Application*, название проекта `TST4VB`.

Сделаем проект стартовым (*Set as StartUp Project*).

Нам нужно добавить ссылку (*reference*) проекта `TST4VB` на проект `BEEPC`. На вкладке *Solution Explorer* должен быть выбран проект `TST4VB`.

Выбираем в меню `Project — Add Reference`. Появится диалог для выбора ссылок на различные виды объектов (рисунок 29).



Component Name	TypeLib Ver...	Path
BEEPC 1.0 Type Library	1.0	C:\AFX104\BEEPC\Debug\BE..
Bined package 8.0 Type Library	8.0	D:\Program Files\Microsoft Vi..
blockdriver 1.0 Type Library	1.0	D:\WINDOWS\system32\hot..
brp_sysinfo 1.0 Type Library	1.0	D:\WINDOWS\PCHealth\Help.
CAPICOM v2.1 Type Library	2.1	D:\Program Files\Common Fil..
CatalogServer 1.0 Type Library	1.0	D:\WINDOWS\system32\cats.
CMProps 1.0 Type Library	1.0	D:\WINDOWS\system32\cmp.
COLBCAT 1.0 Type Library	1.0	D:\WINDOWS\system32\clbc.
coloader 8.0 Type Library	8.0	D:\Program Files\Common Fil..

Рисунок 29 — 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()`:

```
Module Module1
    Sub Main()
        ' создаем объект
        Dim B As New BEEPCLib.Beeper
        ' устанавливаем свойство
        B.Tone = 400
        ' 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`).

Тестовый проект на C++

Добавляем еще один тестовый проект TST104, язык C++, консольное приложение *Win32*, платформа Win32 или x86 в зависимости от того, что есть. Делаем проект стартовым.

Содержание функции main() копирует содержание функций main() и main_b() проекта AFX103. Но, поскольку нам теперь действительно недоступны ни интерфейс, ни кокласс, ни описания идентификаторов, эти сведения нужно получить из библиотеки типов, которая сформирована при компиляции проекта BEEPC.

Эти сведения импортируются с помощью директивы import:

```
#import "C:\AFX104\BEEPC\Debug\BEEPC.tlb" no_namespace named_guids
```

Параметрами является путь к библиотеке типов и флаги. Нам потребуется два флага: no_namespace и named_guids. Первый флаг помещает определения COM-сервера в пространство имен проекта. Второй флаг указывает компилятору определить константы типа CLSID_XXX, IID_XXX.

При включении директивы import в проект добавятся два файла типов .tli и .tlh, являющиеся заместителями библиотеки типов.

Кроме того, вместо указателя на интерфейс следует использовать так называемый smart-указатель (*умный* или *специальный* указатель).

Это обертка вокруг указателя, которая сама выполняет разную полезную работу, например, создание объекта и получение интерфейса, освобождение указателя. Объявляется он следующим образом:

```
    IBeeperPtr pBeep;
```

Здесь IBeeperPtr определяется в файле типа .tlh. Имя типа складывается из имени интерфейса и Ptr (от Pointer, указатель).

Создание объекта выполняется в этом случае при помощи метода CreateInstance(), который есть у smart-указателя. Параметров у этой функции меньше и использовать ее получается проще.

После использования smart-указателя его нужно обязательно установить в NULL или 0:

```
    pBeep->Beep();
    pBeep = NULL;
```

Компилируем, убеждаемся в отсутствии ошибок.

Запускаем этот проект и убеждаемся, что метод Beep() вызывается.

Практическая работа AFX105

В этой работе мы исследуем технологию COM-объектов с подключениями. Такие объекты наследуют также так называемый *исходящий* интерфейс. Клиенты объекта с подключениями *подписываются* на эти интерфейсы, и получают уведомления о событиях в объекте. Ключевые слова — *connection point*. Мы создадим объект `Beeper`, который будет генерировать событие `Beeped`, и клиента, который это событие получит.

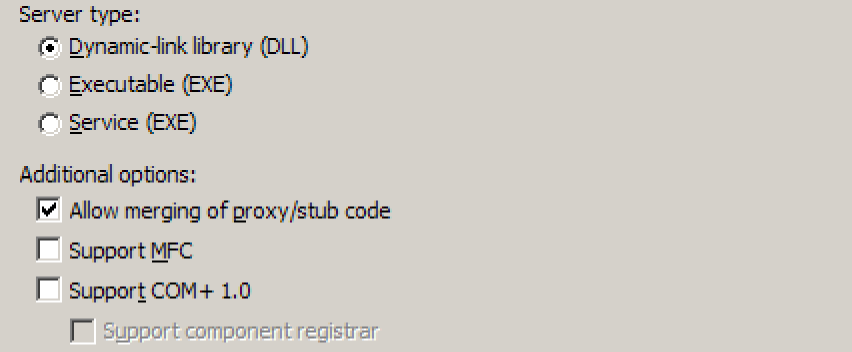
Открываем MSVS.

Создаем новый проект C++, раздел *ATL*, шаблон *ATL Project*.

В мастере *ATL Project Wizard* вводим:

- имя проекта `BEEPE` ,
- расположение проекта `C:\` ,
- имя решения (*solution*) `AFX105` .

Далее на последнем шаге мастера нужно выбрать флажок присоединения проекта прокси и заглушки к основному проекту (рисунок 30).

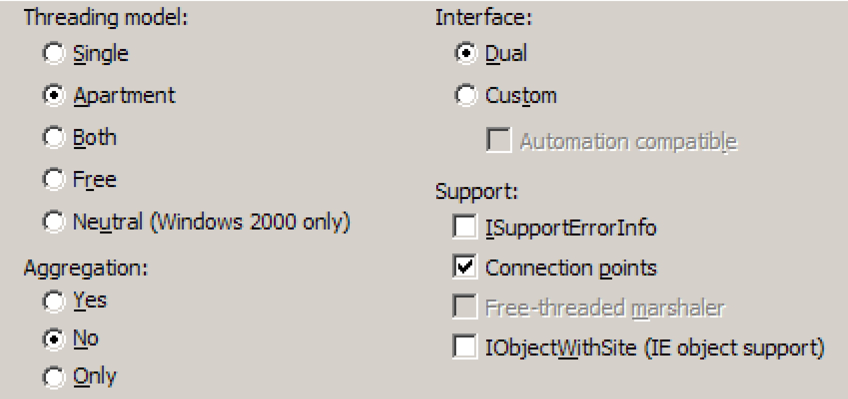


Server type:
- ● Dynamic-link library (DLL)
- ○ Executable (EXE)
- ○ Service (EXE)

Additional options:
- ☑ Allow merging of proxy/stub code
- ☐ Support MFC
- ☐ Support COM+ 1.0
  - ☐ Support component registrar

Рисунок 30 — Присоединение проекта прокси и заглушки

После создания проекта добавляем в него COM-объект при помощи меню `Project — Add Class`. Шаблон объекта *ATL Simple Object*. Короткое имя объекта `Beeper`. На втором шаге мастера создания объекта выбираем объект, поддерживающий точки подключения (рисунок 31).



Threading model:
- ○ Single
- ● Apartment
- ○ Both
- ○ Free
- ○ Neutral (Windows 2000 only)

Aggregation:
- ○ Yes
- ● No
- ○ Only

Interface:
- ● Dual
- ○ Custom
  - ☐ Automation compatible

Support:
- ☐ ISupportErrorInfo
- ☑ Connection points
- ☐ Free-threaded marshaler
- ☐ IObjectWithSite (IE object support)

Рисунок 31 — Создание объекта с подключениями

Откроем файл библиотеки типов BEEPE.idl.

```
import "oaidl.idl";
import "ocidl.idl";
[
    object,
    uuid(7CEA888E-4127-4402-802A-C75E95770CFF),
    dual,
    nonextensible,
    helpstring("IBeeper Interface"),
    pointer_default(unique)
]
interface IBeeper : IDispatch {
};
[
    uuid(F4FEBF3E-85B2-4B82-B31B-A5EC1850A37D),
    version(1.0),
    helpstring("BEEPE 1.0 Type Library")
]
library BEEPELib
{
    importlib("stdole2.tlb");
    [
        uuid(92F7AC9F-07E6-4DE3-891E-784713C81413),
        helpstring("_IBeeperEvents Interface")
    ]
    dispinterface _IBeeperEvents {
        properties:
        methods:
    };
    [
        uuid(326F883A-92B2-4C72-9718-32F017AE2431),
        helpstring("Beeper Class")
    ]
    coclass Beeper {
        [default] interface IBeeper;
        [default, source] dispinterface _IBeeperEvents;
    };
};
```

Обратим внимание на интерфейс _IBeeperEvents, который описывает исходящий интерфейс объекта Beeper. В коклассе этот интерфейс отмечен ключевым словом source.

Далее обычным образом добавляем метод Beep, строка подсказки «Beeps», и свойство Tone, строка подсказки «Returns/Sets Tone».

В описании кокласса в модуле Beeper.h добавляем раздел private и объявляем в нем переменную m_tone. В конструкторе присваиваем переменной m_tone начальное значение.

Для реализации методов в модуле Beeper.h нужно подключить библиотеку <stdio.h>. Метод Beep реализуется обычным образом, выводит в консоль строку "BEEPE::Beep: n." Методы свойства Tone особенностей не имеют и реализуются обычным образом.

Компилируем проект, убеждаемся, что DLL успешно создается.

Так, как описано в работе AFX104, добавляем новый проект на языке *Visual Basic*, название проекта TST5VB. Делаем тестовый проект стартовым. Подключаем к проекту библиотеку типов проекта BEEPE, описываем переменную для объекта, убеждаемся, что объект создается, свойство и метод объекта вызываются.

Теперь перейдем к собственно созданию структур, необходимых для уведомления о событиях объекта Beeper.

Сначала нужно добавить метод Beeped в интерфейс _IBeeperEvents.

Делается это обычным образом, с помощью меню Add — Add Method, применяемым к интерфейсу на вкладке *Class View* (рисунок 32).

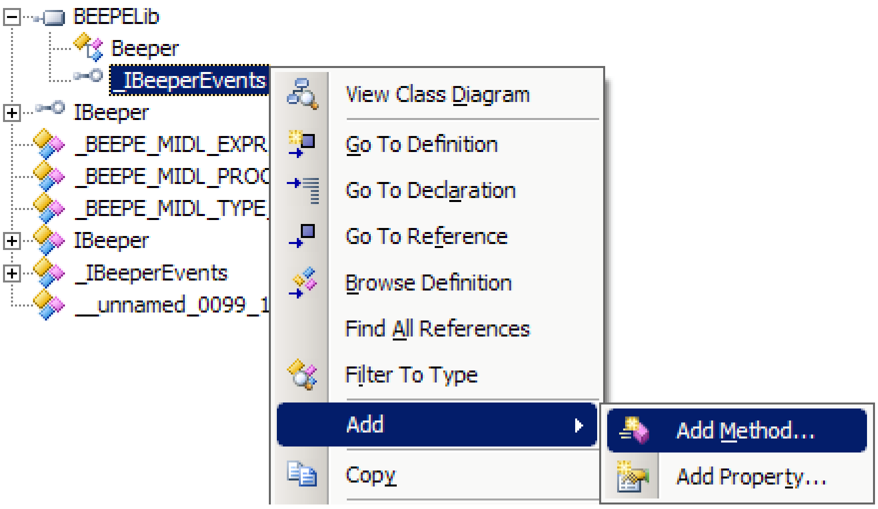


Рисунок 32 — Добавление метода исходящего интерфейса

Название метода, как было сказано, Beeped, параметров нет. Результат в библиотеке типов следующий (модуль BEEPE.idl):

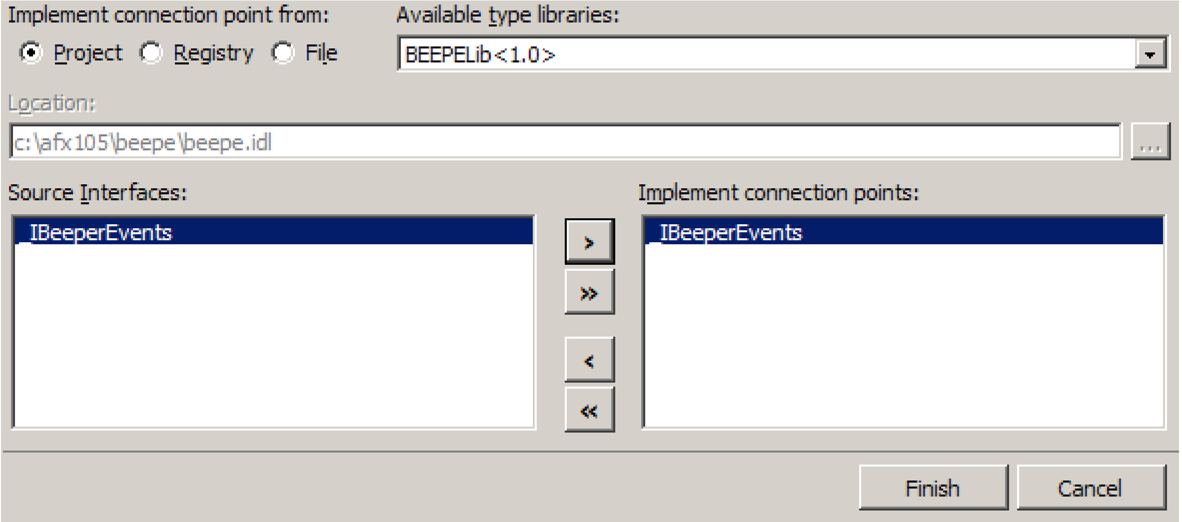
```
    dispinterface _IBeeperEvents {
        properties:
        methods:
            [id(1), helpstring("Beeped")] HRESULT Beeped(void);
    };
```

В проекте есть модуль _IBeeperEvents_CP.h. Откроем его.

```
#pragma once
template <class T>
class CProxy_IBeeperEvents : public IConnectionPointImpl<T, &__uuidof(
_IBeeperEvents), CComDynamicUnkArray> {
    //Warning this class will  be regenerated by the wizard.
public:
};
```

Здесь описывается шаблон класса CProxy_IBeeperEvents, который реализует механизм рассылки событий.

Теперь необходимо реализовать интерфейс _IBeeperEvents. Это сделает мастер *Implement Connection Point Wizard*. Чтобы его вызвать, нужно на вкладке *Class View* выбрать кокласс CBeeper и в контекстном меню выбрать пункт Add — Add Connection Point… (рисунок 33).



Implement connection point from:
● Project ○ Registry ○ File

Available type libraries: BEEPELib<1.0>

Location: c:\afx105\beepe\beepe.idl

Source Interfaces: _IBeeperEvents

Implement connection points: _IBeeperEvents

Рисунок 33 — Мастер добавления точки соединения

Исходящий интерфейс _IBeeperEvents нужно добавить из левого списка в правый список при помощи кнопки ">" и нажать *Finish*.

В результате в модуле _IBeeperEvents_CP.h появится описание метода Fire_Beeped(), который рассылает событие всем подписавшимся:

```
HRESULT Fire_Beeped() {
    HRESULT hr = S_OK;
    T * pThis = static_cast<T *>(this);
    int cConns = m_vec.GetSize();
    for (int iConn = 0; iConn < cConns; iConn++) {
        pThis->Lock();
        CComPtr<IUnknown> punkConn = m_vec.GetAt(iConn);
        pThis->Unlock();
        IDispatch * pConn = static_cast<IDispatch *>(punkConn.p);
        if (pConn) {
            CComVariant varResult;
            DISPPARAMS params = { NULL, NULL, 0, 0 };
            hr = pConn->Invoke(1, IID_NULL, LOCALE_USER_DEFAULT,
                DISPATCH_METHOD, &params, &varResult, NULL, NULL);
        }
    }
    return hr;
}
```

Понять, что здесь написано, можно только после изучения технологии OLE Automation (автоматизация). Смысл такой: получить вектор подписавшихся клиентов m_vec, узнать количество cConns, для каждого клиента вызвать методом Invoke метод исходящего интерфейса номер 1.

Подписавшиеся клиенты обязаны реализовать исходящий интерфейс как интерфейс диспетчеризации IDispatch, а метод Invoke этого интерфейса вызывает метод по его номеру. У нас всего то один метод.

Все детали этого механизма достаточно сложны. В основе всего лежит интерфейс IConnectionPoint. Откроем модуль Beeper.h.

```
class ATL_NO_VTABLE CBeeper :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CBeeper, &CLSID_Beeper>,
    public IConnectionPointContainerImpl<CBeeper>,
    public CProxy_IBeeperEvents<CBeeper>,
    public IDispatchImpl<IBeeper, &IID_IBeeper, &LIBID_BEEPELib, 1, 0>
```

Щелкните правой кнопкой мыши на IConnectionPointContainerImpl и выберите *Go To Definition*. В открывшемся модуле найдите название интерфейса IConnectionPoint и таким же образом перейдите на его описание:

```
    MIDL_INTERFACE("B196B286-BAB4-101A-B69C-00AA00341D07")
    IConnectionPoint : public IUnknown {
    public:
        virtual HRESULT STDMETHODCALLTYPE GetConnectionInterface(
            /* [out] */ IID *pIID) = 0;

        virtual HRESULT STDMETHODCALLTYPE GetConnectionPointContainer(
            /* [out] */ IConnectionPointContainer **ppCPC) = 0;

        virtual HRESULT STDMETHODCALLTYPE Advise(
            /* [in]  */ IUnknown *pUnkSink,
            /* [out] */ DWORD *pdwCookie) = 0;

        virtual HRESULT STDMETHODCALLTYPE Unadvise(
            /* [in]  */ DWORD dwCookie) = 0;

        virtual HRESULT STDMETHODCALLTYPE EnumConnections(
            /* [out] */ IEnumConnections **ppEnum) = 0;
    };
```

Это стандартный интерфейс, который позволяет клиенту соединяться и разъединяться с объектом-источником. Метод Advise используется клиентом для соединения, метод Unadvise — для разъединения.

Всего в технологии задействовано 4 интерфейса:

IConnectionPoint
IConnectionPointContainer
IEnumConnectionPoints
IEnumConnections

Интерфейс IConnectionPointContainer позволяет клиенту просматривать все соединяемые объекты контейнера.

Интерфейс IEnumConnectionPoints перечисляет все точки соединения.

Интерфейс IEnumConnections перечисляет подключенных клиентов.

Рассмотрение этих интерфейсов выходит за рамки целей изучения.

Нам остается только сгенерировать событие на стороне сервера, и получить его на стороне клиента.

Переходим в модуль Beeper.cpp.

В методе Beep вызываем событие Beeped:

```
#include "stdafx.h"
#include "Beeper.h"
#include <stdio.h>
// CBeeper
STDMETHODIMP CBeeper::Beep(void) {
    printf("BEEPE::Beep: %d.\n\n", m_tone);
    Fire_Beeped();
    return S_OK;
}
```

Переходим в тестовый проект TST5VB.

Вероятно, нужно заново вызвать диалог *References* и установить связь с библиотекой типов проекта BEEPE.

Объявляем переменную, которая будет связана с событием:

```
Module Module1
    Private WithEvents BE As New BEEPELib.Beeper
    Sub Main()
        . . .
    End Sub
End Module
```

Ключевое слово WithEvents здесь создает объект с подключением.

Описываем действия с объектом BE:

```
    Sub Main()
        Dim B As New BEEPELib.Beeper
        B.Tone = 400
        B.Beep()
        BE.Tone = 100
        BE.Beep()
    End Sub
```

В левом списке модуля выбираем объект BE, в правом списке выбираем событие Beeped (рисунок 34).

```
Module Module1
```

Рисунок 34 — Выбор объекта и его события в модуле Visual Basic

Появится реализация исходящего интерфейса, в которую мы вписываем вывод на консоль сообщения "Beeped":

```
    Private Sub BE_Beeped() Handles BE.Beeped
        Console.WriteLine("Beeped" & vbNewLine)
    End Sub
```

Компилируем, запускаем, убеждаемся, что событие происходит.

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 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 программному обеспечению 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 кокласcу. 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.

Компонент, не установленный в приложение COM+, называется несконфигурированным компонентом (*unconfigured*).

После того, как компонент был установлен в приложение COM+, он становится сконфигурированным компонентом (*configured component*).

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

Существует несколько типов 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 35).

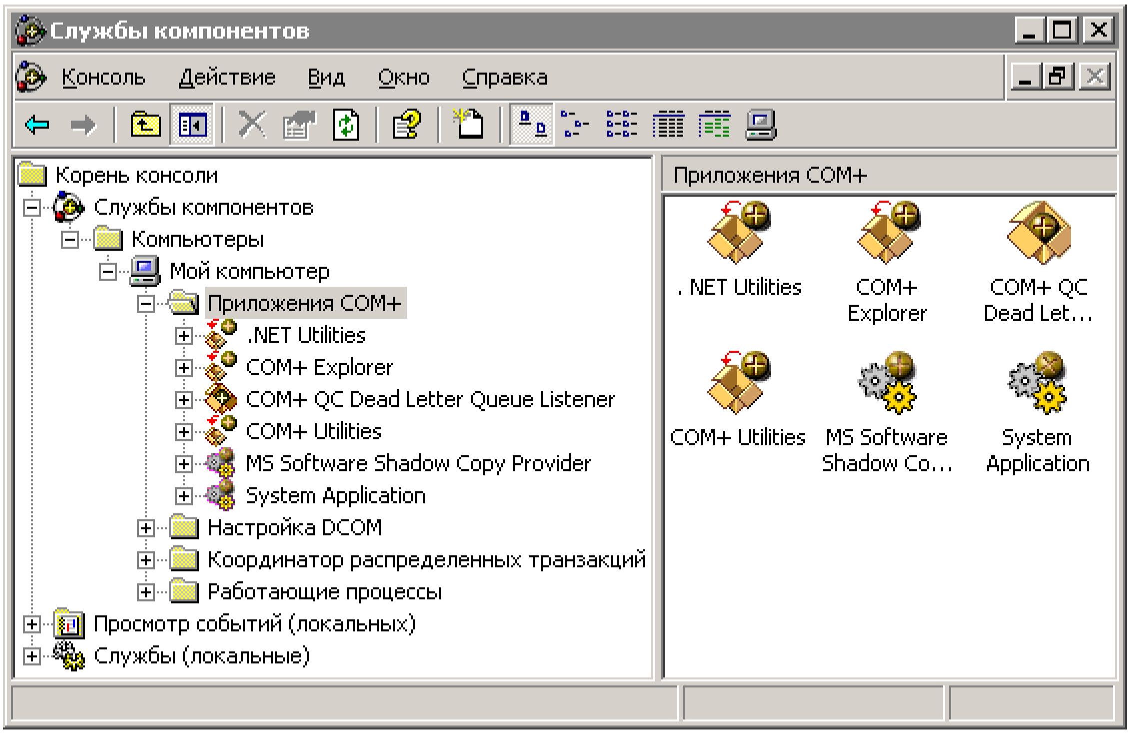


Рисунок 35 — Консоль «Службы компонентов»

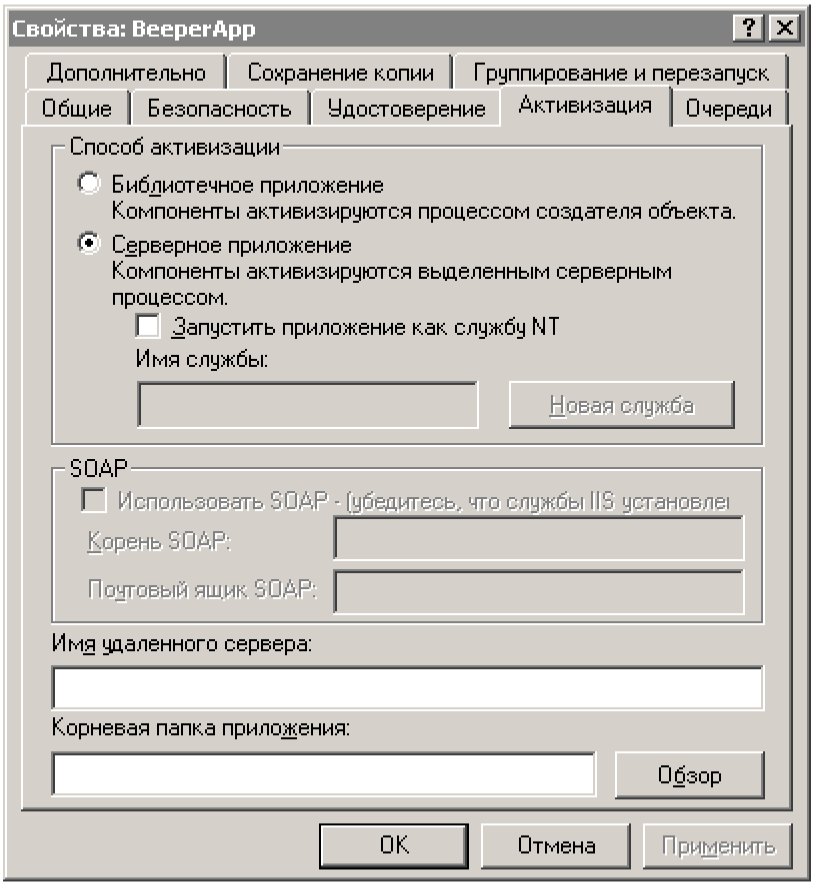
Не следует путать сервисы COM+ с сервисами NT операционной системы. Сервисы COM+ — это обеспечиваемые свойства среды, в которой работают компоненты COM+. К этим свойствам относят безопасность, согласованность, транзакции, активизацию и другие.

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 36).



Способ активизации

○ Библиотечное приложение
Компоненты активизируются процессом создателя объекта.

● Серверное приложение
Компоненты активизируются выделенным серверным процессом.

☐ Запустить приложение как службу NT

Имя службы:

SOAP

☐ Использовать SOAP - (убедитесь, что службы IIS установле

Корень SOAP:

Почтовый ящик SOAP:

Имя удаленного сервера:

Корневая папка приложения:

Рисунок 36 — Атрибуты компонента

Значения атрибутов хранятся в конфигурационной базе данных, называемой каталогом (рисунок 37).

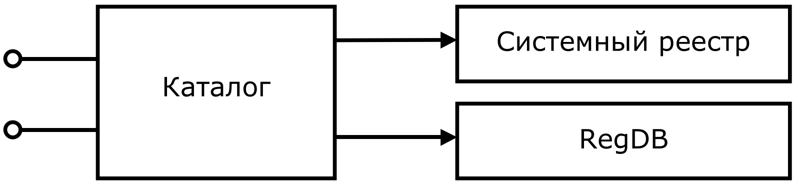


Рисунок 37 — Каталог COM+

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 38).

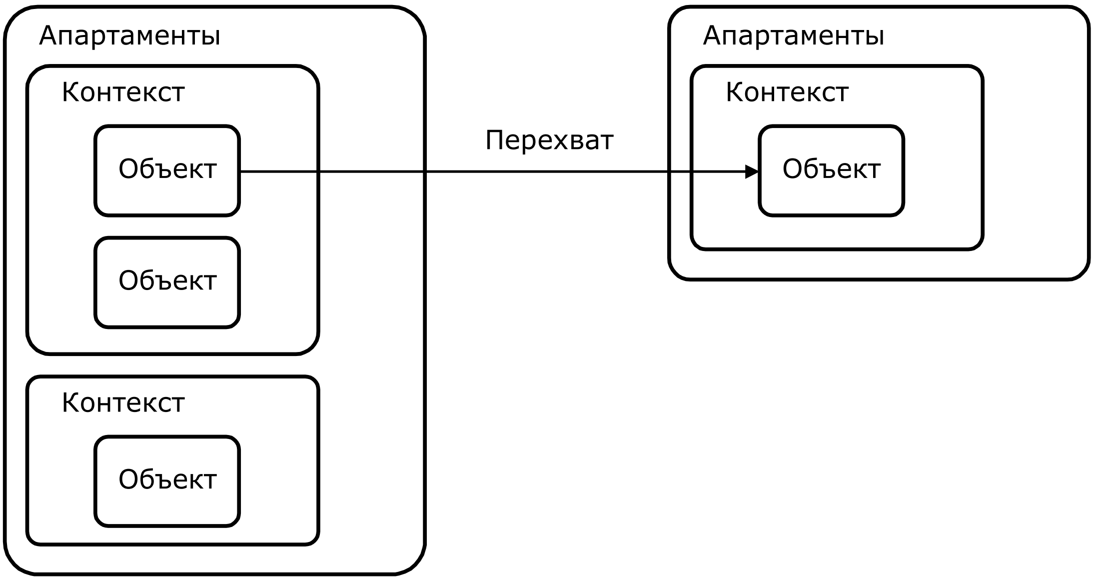


Рисунок 38 — Объект, контекст, апартаменты

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.

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.

Жизненный цикл объекта показан на рисунке 39.

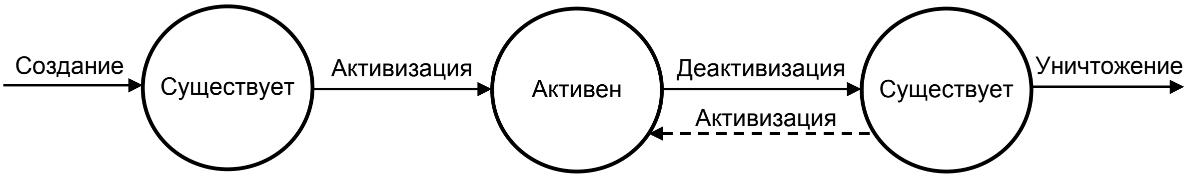


Рисунок 39 — Жизненный цикл объекта COM+

Практическая работа AFX201

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 AFX104». 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 (BEEP Plus), наименование кокласса Beeper, и сервер располагается в папке C:\BEEPP.

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 *Готов*» (рисунок 40).

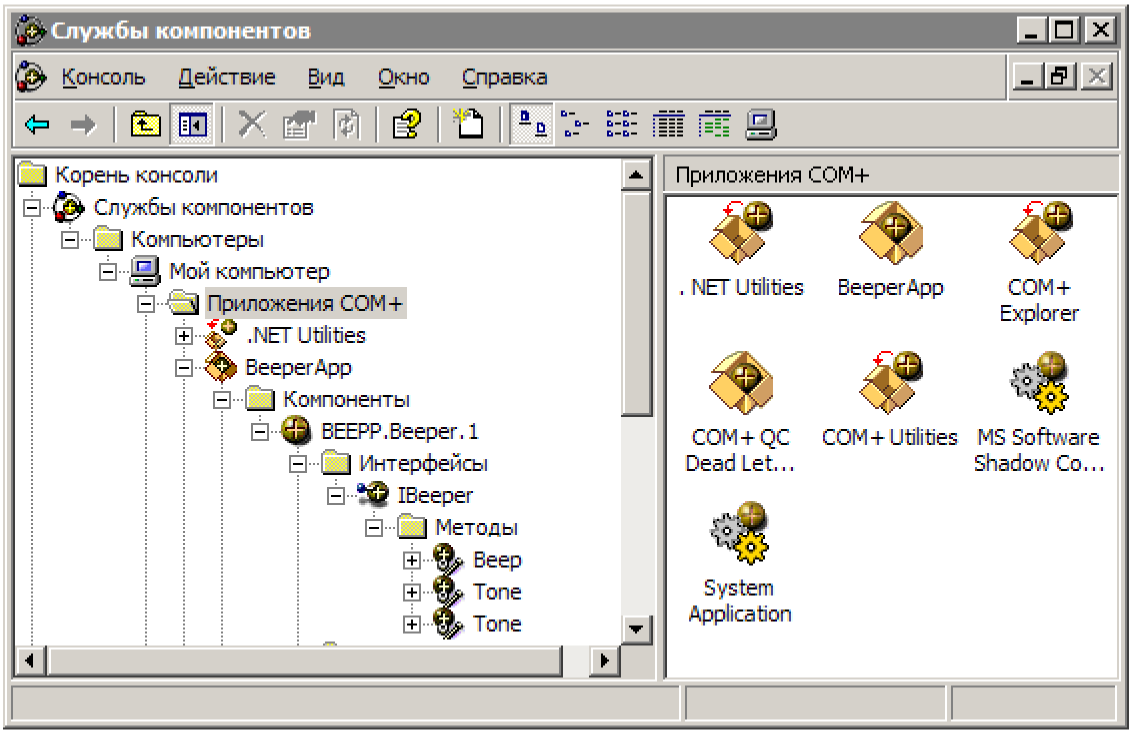


Рисунок 40 — Установка приложения COM+

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 AFX104. 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. Одновременно смотрите на шарик приложения COM+. Наслаждайтесь эффектом вращения шарика COM+, который показывает, что система COM+ находится в действии.

Еще лучше в методе Beep вставить код, выводящий диалоговое окно сообщения. Тогда после запуска тестирующего приложения выполнение приостановится до момента закрытия этого окна.

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 (в это время он должен находиться в пуле объектов).

Кроме того, открыв диалог атрибутов объекта (рисунок 41, 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].

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
(сайт: http:\\revol.ponocom.ru\, e-mail: 245-59@mail.ru)
Введение в ActiveX. Часть 1. Инфраструктура COM
Учебно-методическое пособие

Отпечатано с готового оригинал-макета
Издательство ОТИ НИЯУ МИФИ, 2016
Тираж 10 экз.